Посвящается Дэйву Катлеру — отцу ядра Windows

Mark E. Russinovich
David A. Solomon

Microsoft® Windows® Internals Fourth Edition:

Windows Server™ 2003, Windows XP, and Windows 2000

Microsoft® Press

Марк Руссинович
Дэвид Соломон

Внутреннее устройство Microsoft® Windows®:

Windows Server™ 2003, Windows XP и Windows 2000

4-е издание

МАСТЕР-КЛАСС

РУССКАЯ РЕДАКЦИЯ ПИТЕР®

Москва • Санкт-Петербург • Нижний Новгород • Воронеж
Новосибирск • Ростов-на-Дону • Екатеринбург • Самара
Киев • Харьков • Минск
2008

УДК 004.45
ББК 32.973.26–018.2
Р89

Руссинович М. и Соломон Д.

Р89 Внутреннее устройство Microsoft Windows: Windows Server 2003, Windows XP и Windows 2000. Мастер-класс. / Пер. с англ. — 4-е изд. — М.: Издательство «Русская Редакция»; СПб.: Питер, 2008. — 992 стр.: ил.

ISBN 978–5–469–01174–3 («Питер»)
ISBN 978–5–7502–0085–6 («Русская Редакция»)

Книга посвящена внутреннему устройству и алгоритмам работы основных компонентов операционной системы Microsoft Windows — Windows Server 2003, Windows XP и Windows 2000 — и файловой системы NTFS. Детально рассмотрены системные механизмы: диспетчеризация ловушек и прерываний, DPC, APC, LPC, RPC, синхронизация, системные рабочие потоки, глобальные флаги и др. Также описываются все этапы загрузки операционной системы и завершения ее работы. В четвертом издании книги больше внимания уделяется глубокому анализу и устранению проблем, из-за которых происходит крах операционной системы или из-за которых ее не удается загрузить. Кроме того, рассматриваются детали реализации поддержки аппаратных платформ AMD x64 и Intel IA64. Книга состоит из 14 глав, словаря терминов и предметного указателя. Книга предназначена системным администраторам, разработчикам серьезных приложений и всем, кто хочет понять, как устроена операционная система Windows.

Названия всех команд, диалоговых окон и других интерфейсных элементов операционной системы приведены как на английском языке, так и на русском.

УДК 004.45
ББК 32.973.26-018.2

Подготовлено к печати издательско-торговым домом «Русская Редакция» по лицензионному договору с Microsoft Corporation, Редмонд, Вашингтон, США.

ISBN 0–7356–1917–4 (англ.)
ISBN 978–5–469–01174—3 («Питер»)
ISBN 978–5–7502–0085–6 («Русская Редакция»)

Оглавление

Предыстория

Я вновь признателен Дэвиду Соломону (David Solomon) и Марку Руссиновичу (Mark Russinovich) за то, что они предоставили мне возможность сказать несколько слов о новом издании их книги «Внутреннее устройство Microsoft Windows». Прошло уже более трех лет с момента выхода последнего издания этой книги, и за это время на свет появились два выпуска операционной системы Windows — очень значимые обновления клиентской и серверной систем.

Перед авторами стояли две задачи, которые постоянно усложняются: отслеживание эволюционного развития системы Microsoft Windows NT и документирование того, как менялась реализация ее компонентов в каждой версии. В этом смысле авторы проделали просто выдающуюся работу.

(Слева направо) **Дэвид Соломон, Дэвид Катлер и Марк Руссинович**

Впервые я познакомился с Дэвидом Соломоном, когда ему было всего 16 лет, а я работал в Digital Equipment Corporation над операционной системой VMS для VAX. С тех пор он участвовал в разработке операционных систем, а также преподавал в этой области. С Марком Руссиновичем я познакомился позже, но уже задолго до этого был немало наслышан о его глубоких познаниях в области операционных систем. В числе его заслуг версия файловой системы NTFS, которую он заставил работать в Microsoft Windows 98, и «живой» отладчик ядра Windows, позволяющий заглянуть внутрь системы Windows в процессе ее выполнения.

Истоки Windows NT восходят к октябрю 1988 года, когда было решено создать переносимую операционную систему, совместимую с OS/2, поддерживающую POSIX и многопроцессорную обработку, обладающую высокой защищенностью, надежностью и интегрированными средствами работы в сетях. С приходом Windows 3.0 и ее колоссальным успехом заявленные цели несколько изменились: совместимость с OS/2 была перенесена с уровня всей системы на уровень отдельной подсистемы.

Поначалу мы полагали, что сумеем создать Windows NT за пару лет, но в действительности ее первая версия вышла лишь через четыре с половиной года — летом 1993-го. Эта версия поддерживала процессоры Intel i386, Intel i486 и MIPS R4000. Шесть недель спустя мы ввели поддержку и для процессоров Digital Alpha.

Первая версия Windows NT получилась более громоздкой и медленной, чем ожидалось, так что следующей вехой стал проект Daytona (так называется автострада во Флориде). Главной целью в этой версии было уменьшение размера системы, повышение ее быстродействия и, разумеется, надежности. Через полгода после выпуска Windows NT 3.5 осенью 1994-го мы подготовили Windows NT 3.51, которая представляла собой обновленную версию с дополнительной поддержкой процессора IBM PowerPC.

Толчком к созданию следующей версии Windows NT стало желание сделать пользовательский интерфейс, совместимый с Windows 95, и включить технологии Cairo, уже находившиеся в разработке пару лет. На создание этой системы ушло еще два года, и летом 1996 года была представлена Windows NT 4.0.

Название следующей версии NT было изменено на Windows 2000. Она стала последней системой, для которой одновременно выпускались клиентская и серверная версии. Windows 2000 была построена на той же технологии Windows NT, что и предыдущие версии, но обладала новой важной функциональностью, поддерживая, в частности, Active Directory. На разработку Windows 2000 ушло три с половиной года, и на тот период она была самой оптимизированной и наиболее тщательно протестированной версией технологии Windows NT. Windows 2000 стала кульминацией более чем одиннадцатилетних разработок, реализованных на четырех архитектурах.

В конце разработки Windows 2000 мы приступили к работе над амбициозным планом реализации новых версий клиентской и серверной систем, которые должны были предоставить новые, более совершенные возможности потребителям и улучшить характеристики серверов. Но потом стало ясно, что реализация серверных средств привела бы к задержке в реализации клиентских, и поэтому было решено разделить выпуски. В августе 2001 года на свет появились Windows XP Professional и Windows XP Home Edition, а через год с небольшим, в марте 2003 года была выпущена Microsoft Windows Server 2003. Помимо архитектуры Intel x86, эти системы поддерживали Intel IA64, благодаря чему Windows NT впервые вышла на стезю 64-разрядных вычислений.

Эта книга единственная, где так глубоко и полно рассмотрены внутренние структуры и принципы функционирования Windows XP и Windows Server

2003. Кроме того, она предлагает заглянуть в будущее — перевод Windows на 64-разрядные «рельсы», т. е. на ее поддержку архитектур x64 (AMD64) и Intel EM64T, объявленных AMD в 2003 году и Intel в феврале 2004 года соответственно. Выпуск клиентской и серверной версий с полной поддержкой x64 запланирован на первую половину 2005 года, и в этой книге содержится масса информации о внутренних деталях реализации x64-системы.

Архитектура x64 — это начало новой эры для Windows NT в тот момент, когда время архитектуры x86 подходит к концу. Архитектура x64 обеспечивает совместимость с 32-разрядной x86-платформой и предоставляет 64-разрядную адресацию для наиболее требовательных, совершенно новых приложений. Это позволит сохранить инвестиции в 32-разрядное программное обеспечение, в то же время вдохнув новую жизнь в Windows NT на ближайшее десятилетие или даже на более длительный период.

Хотя название NT-системы за последние несколько лет неоднократно менялось, она по-прежнему полностью основана на исходной кодовой базе Windows NT. Но время бежит, появляются новые технологии, и реализация многих внутренних компонентов и функций значительно изменилась. Авторы проделали внушительную работу, собрав столько детальной информации о кодовой базе Windows NT и ее реализациях в разных выпусках на разных платформах, а также создав примеры и утилиты, которые помогают читателю разобраться в том, как работают компоненты и подсистемы Windows. Экземпляр этой книги должен лежать на столе у каждого разработчика серьезного программного обеспечения.

Дэвид Н. Катлер,
заслуженный старший инженер
корпорации Microsoft

Предисловие

Microsoft Windows была частью моей жизни целых 14 лет. За это время — от версии к версии — наша операционная система развивалась вширь и вглубь. Сегодня работа над Windows — один из самых важных и сложных проектов в мире. В выпуске Windows участвуют более 5000 инженеров. Среди пользователей Windows есть представители уже почти всех культур, она используется как на крупных предприятиях, так и маленькими детьми. Пользователи Windows постоянно требуют ее совершенствования практически во всех сферах — от эффективной работы на крупнейших серверах до применения в дошкольном обучении. Windows поставляется в самых разных ипостасях — от встраиваемых версий и выпусков для медиа-центров до редакций для центров обработки данных. Все эти продукты опираются на одни и те же базовые компоненты Windows, которые развиваются и совершенствуются в каждой новой версии.

Это фундаментальная книга о внутреннем устройстве базовых компонентов Windows. Если вы хотите как можно быстрее освоить принципы внутренней работы Windows, тогда эта книга для вас. Освоение всех частей столь основательного продукта — задача устрашающая. Но если вы начнете с базовых концепций системы, сложить фрагменты головоломки воедино будет гораздо легче. С эволюцией самой Windows развивается и эта книга — сейчас публикуется ее четвертое издание. Мы уже давно используем ее для обучения новых сотрудников Microsoft, так что предлагаемые вам материалы проверены на практике.

Если вы вроде меня, значит, вам тоже нравится разбираться в том, как устроены вещи. Чтение книжек типа «как использовать то-то и то-то» меня никогда не удовлетворяло. Когда понимаешь, как именно устроена вещь, пользуешься ею гораздо эффективнее и, честно говоря, с бóльшим удовольствием. Если вас интересует Windows «с изнанки», вы выбрали подходящую книгу.

Дэвид и Марк проделали превосходную работу, написав книгу о технической «изнанке» Windows. А инструменты, которые они предлагают вам, — отличное средство для самостоятельного обучения и диагностики. Прочитав эту книгу, вы будете гораздо лучше понимать, как взаимодействуют между собой различные компоненты и подсистемы, какие усовершенствования внесены в новую версию и как выжать из них максимум возможного.

Это был долгий путь — и он все еще продолжается. Так что открывайте книгу, а заодно и капот, под которым бьется сердце одной из самых потрясающих операционных систем.

Джим Олчин,
вице-президент группы платформ
корпорации Microsoft

Благодарности

В первую очередь мы хотим особо поблагодарить следующих людей.

- **Дэйва Катлера** (Dave Cutler), заслуженного старшего инженера и первого архитектора Microsoft Windows NT. Дэйв разрешил Дэвиду Соломону (David Solomon) доступ к исходному коду и всячески поддерживал его преподавательскую деятельность, посвященную объяснению деталей внутреннего устройства Windows NT, а также его работу над вторым и третьим изданием книги. Помимо рецензирования главы по процессам и потокам, Дэйв ответил на массу вопросов об архитектуре ядра и написал об истории создания Windows для нашей книги.
- **Джима Олчина** (Jim Allchin), нашего главного спонсора, — за предисловие к этой книге и за отстаивание интересов нашего дела в Microsoft.
- **Роба Шорта** (Rob Short), вице-президента, который позаботился о том, чтобы нам предоставили ресурсы и доступ к нужным людям.

Мы также выражаем признательность двум разработчикам из отдела Windows за подготовку новых материалов, включенных в это издание:

- **Адриану Маринеску** (Adrian Marinescu), который написал заметно разросшийся раздел по диспетчеру куч в главе, где рассматривается диспетчер памяти.
- **Самеру Арафеху** (Samer Arafeh), который предоставил материалы по Wow64.

Спасибо нашему старому приятелю, Джеффри Рихтеру (Jeffrey Richter), с которым мы часто вместе обедаем, за врезку «Как насчет .NET и WinFX?» в главе 1 и за постоянное напоминание о том, как мало людей, по-настоящему интересующихся тем, о чем мы говорим в своей книге.

В этой книге не было бы такой глубины и точности изложения технических сведений без поддержки, замечаний и предложений ключевых членов команды разработчиков Microsoft Windows. Вот эти люди:

Murali Brahmadesam	Pat Hoffer	Daniel Pravat
Molly Brown	Anthony Jones	Dragos Sambotin
Duncan Bryce	Tom Jones	Jon Schwartz
Daniel Bucherer	Joseph Joy	Rob Short
Neal Christian	Shreeniwas Kelkar	Paul Sliwowicz
Neill Clift	Connie La Chasse	Chittur Subbaraman
Mike Danseglio	Mike Lai	Cristian Teodorescu
Joseph Davies	Paul Leach	Andre Vachon
Cenk Ergan	Gerald Maffeo	Landy Wang
Tom Fout	Aaron Margosis	Richard Ward
Nar Ganapathy	Iain McDonald	Brad Waters
David Golds	Kamen Moutafov	Bruce Worthington
Robert Gu	Adi Oltean	Mark Zbikowsk
Jeff Hamblin	Vince Orgovan	Khawar Zuberi

Были и другие, кто отвечал на наши вопросы в коридорах или кафетериях, — если мы вас пропустили, пожалуйста, простите нас!

Мы также выражаем благодарность Джейми Ханрахан (Jamie Hanrahan) из Azius Developer Training (*www.azius.com*), которая в соавторстве с Дэвидом подготовила учебный курс по внутренней архитектуре исходной версии Windows. На основе этого курса было написано второе издание этой книги. Джейми, у которой настоящий талант доходчиво объяснять сложнейшие вещи, предоставила нам отдельные материалы, а также ряд схем и иллюстраций.

Спасибо Дэйву Проберту (Dave Probert) за то, что разместил в сети наши черновые материалы для распространения среди рецензентов внутри Microsoft.

Благодарим Джонатана Славза (Jonathan Sloves) из AMD, с помощью которого нам предоставили тестовые системы AMD64; они очень помогли нам в написании материалов по 64-разрядной архитектуре и в переносе ряда утилит Sysinternals на платформу x64.

Наконец, мы хотим выразить благодарность следующим сотрудникам Microsoft Press за их вклад в эту книгу.

- **Робину Ван-Штеенбергу** (Robin van Steenburgh), рецензенту издательства, за терпение в работе с нами над этим проектом.
- **Салли Стикни** (Sally Stickney), которая на первых порах по-прежнему была редактором нашего проекта, но потом ее закрутил водоворот административных дел. Мы очень скучали без вас!
- **Валери Вулли** (Valerie Woolley), которая приняла бразды правления от Салли и стала нашим новым редактором проекта. Вы замечательная и не такая резкая, как Салли!
- **Роджеру Лебланку** (Roger LeBlanc), который одолел все главы и сумел сократить в них текст, найти несогласованности и вообще довести нашу рукопись до высоких стандартов Microsoft Press.

Дэвид Соломон и Марк Руссинович
сентябрь 2004 г.

Введение

Четвертое издание этой книги ориентировано на квалифицированных специалистов (программистов, разработчиков и системных администраторов), желающих разобраться в принципах внутренней работы основных компонентов операционных систем Microsoft Windows 2000, Microsoft Windows XP и Microsoft Windows Server 2003. Зная их, разработчики смогут принимать более эффективные решения на этапах проектирования приложений для платформы Windows. Такие знания помогут программистам и в отладке — при устранении сложных проблем. Информация, изложенная в книге, будет также полезна системным администраторам: понимание того, как устроена и работает операционная система, упростит им оптимизацию своих систем и устранение неполадок в случае каких-либо сбоев. Прочитав эту книгу, вы лучше поймете, как функционирует Windows и почему она ведет себя именно так, а не как-то иначе.

Структура книги

Первые две главы закладывают фундамент, вводя термины и концепции, используемые во всей книге. Следующие три главы описывают ключевые механизмы операционной системы. В следующих восьми главах детально рассматриваются базовые компоненты Windows — процессы, потоки и задания, управление памятью, защита, подсистема ввода-вывода, управление внешней памятью, диспетчер кэша, файловые системы и поддержка сетей. Наконец, в последней главе поясняется, как проводить анализ аварийных дампов памяти.

История написания книги

Это четвертое издание книги, которая изначально называлась «Inside Windows NT» (Microsoft Press, 1992) и была написана Хелен Кастер (Helen Custer) еще до выпуска Microsoft Windows NT 3.1. Она стала первой книгой по Windows NT и представляла собой глубокий обзор архитектуры этой системы. Второе издание, «Inside Windows NT» (Microsoft Press, 1998), было написано Дэвидом Соломоном. В него вошла новая информация по Windows NT 4.0, а сама книга стала гораздо более глубокой. Третье издание, «Inside Windows 2000» (Microsoft Press, 2000), было подготовлено Дэвидом Соломоном и Марком Руссиновичем. В нем появилось много новых тематических разделов, в том числе по этапам загрузки и завершения работы системы, внутреннему устройству сервисов и реестра, по драйверам файловых систем, поддержке

сетей, а также по новой функциональности ядра Windows 2000, например модели WDM, Plug and Play, WMI, шифрованию, Terminal Services и др.

Особенности четвертого издания

Новое издание дополнено информацией об изменениях в ядре, которые были внесены в Windows XP и Windows Server 2003, в том числе касающихся поддержки 64-разрядных систем. Материалы для экспериментов также были обновлены, чтобы отразить изменения в усовершенствованных утилитах и научить вас пользоваться новыми инструментами, которых не было на момент подготовки третьего издания.

Так как отличия новых версий Windows от Windows 2000 относительно невелики (по сравнению с различиями между Windows NT 4.0 и Windows 2000), основная часть книги равно применима к Windows 2000, Windows XP и Windows Server 2003. Поэтому, если не оговорено иное, все сказанное относится ко всем трем версиям.

Инструменты для проведения экспериментов

Даже без доступа к исходному коду существующие инструменты вроде отладчика ядра позволяют многое прояснить во внутреннем устройстве Windows. В том месте, где для демонстрации какого-либо аспекта поведения Windows используется тот или иной инструмент, во врезке «Эксперимент» даются инструкции по его применению. Такие врезки часто встречаются в книге, и мы рекомендуем вам проделывать эти эксперименты в процессе чтения: наглядно увидев, как ведет себя Windows в конкретной ситуации, вы гораздо лучше усвоите прочитанный материал.

Тематика, не рассматриваемая в книге

Windows — большая и сложная операционная система. Нельзя объять необъятное, и поэтому основное внимание в книге уделяется только базовым системным компонентам. Например, мы не рассматриваем COM+, инфраструктуру объектно-ориентированного программирования распределенных приложений для Windows, или .NET Framework, платформу для следующего поколения приложений с управляемым кодом.

Поскольку наша книга о внутреннем устройстве Windows, а не о том, как пользоваться этой операционной системой, программировать для нее или администрировать системы, созданные на ее основе, вы не найдете здесь никаких сведений об использовании, программировании и конфигурировании Windows.

Подводные камни

В книге описываются недокументированные внутренние структуры и функции ядра, архитектура и различные аспекты внутренней работы Windows, и часть таких структур и функций может измениться в следующем выпуске этой операционной системы. (Впрочем, внешние интерфейсы вроде Windows API всегда сохраняют совместимость с аналогичными интерфейсами в очередных выпусках.)

Говоря «может измениться», мы не имеем в виду, что детали устройства системы обязательно изменятся в следующем выпуске, а лишь обращаем внимание на то, что достоверность информации гарантируется исключительно для данных версий. Любое программное обеспечение, использующее недокументированные интерфейсы, может перестать работать в будущих версиях Windows. Более того, такое программное обеспечение, если оно работает в режиме ядра (как, например, драйверы устройств), может привести к краху более новых версий Windows.

Техническая поддержка

Мы приложили максимум усилий, чтобы не допустить неточностей и ошибок в книге. Если у вас возникнут какие-либо проблемы или вопросы, пожалуйста, обращайтесь по адресам, указанным в следующих двух разделах.

От авторов

Эта книга отнюдь не совершенна. Несомненно в ней есть какие-то неточности; может быть, мы упустили что-то важное. Если вы найдете то, что считаете ошибочным, или если вы сочтете, что в книгу следует включить дополнительный материал, пожалуйста, пошлите свое сообщение по адресу *windowsinternals@sysinternals.com*. Обновления и исправления будут выкладываться на страницу *www.sysinternals.com/windowsinternals*.*

От Microsoft Press

Microsoft публикует исправления к книгам по адресу *http://www.microsoft.com/learning/support*. Для прямого подключения к Microsoft Learning Knowledge Base и ввода запроса по проблеме, с которой вы столкнулись, заходите на страницу *http://www.microsoft.com/learning/support/search.asp*.

* В переводе учтены исправления, опубликованные на этой Web-странице, по состоянию на 1 июля 2005 года. — *Прим. перев.*

Вы можете не только напрямую обращаться к авторам, но и посылать свои комментарии, вопросы или соображения по этой книге одним из перечисленных ниже способов.

Обычная почта:

Microsoft Press
Attn: Windows Internals Editor
One Microsoft Way
Redmond, WA 98052-6399

Электронная почта:
mspinput@microsoft.com

Пожалуйста, заметьте, что эти адреса не предназначены для технической поддержки программных продуктов. Информацию о том, как получить техническую поддержку по Microsoft Windows, вы найдете по ссылке *www.microsoft.com/windows* или *support.microsoft.com/support*.

ГЛАВА 1

Концепции и инструменты

В этой главе мы познакомим вас с основными концепциями и терминами операционной системы Microsoft Windows, которые будут использоваться в последующих главах, в том числе с Windows API, процессами, потоками, виртуальной памятью, режимом ядра и пользовательским режимом, объектами, описателями (handles), защитой, реестром. Мы также расскажем об инструментах, с помощью которых вы сможете исследовать внутреннее устройство Windows. К ним относятся, например, отладчик ядра, оснастка Performance и важнейшие утилиты с сайта *www.sysinternals.com*. Кроме того, мы поясним, как пользоваться Windows Device Driver Kit (DDK) и Platform Software Development Kit (SDK) в качестве источника дополнительной информации о внутреннем устройстве Windows.

Вы должны хорошо понимать все, что написано в этой главе, — в остальной части книги мы предполагаем именно так.

Версии операционных систем Windows

Эта книга охватывает три последние версии операционной системы Microsoft Windows, основанные на кодовой базе Windows NT: Windows 2000, Windows XP (32- и 64-разрядные версии) и Windows Server 2003 (32- и 64-разрядные версии). Текст относится ко всем трем версиям, если не оговорено иное. В таблице 1-1 перечислены выпуски кодовой базы Windows NT, номера версий и названия продуктов.

Таблица 1-1. *Выпуски операционной системы Windows*

Название продукта	Номер версии	Дата выпуска
Windows NT 3.1	3.1	Июль 1993 г.
Windows NT 3.5	3.5	Сентябрь 1994 г.
Windows NT 3.51	3.51	Май 1995 г.
Windows NT 4.0	4.0	Июль 1996 г.
Windows 2000	5.0	Декабрь 1999 г.
Windows XP	5.1	Август 2001 г.
Windows Server 2003	5.2	Март 2003 г.

Windows NT и Windows 95

При первом выпуске Windows NT компания Microsoft дала ясно понять, что это долгосрочная замена Windows 95 (и ее последующих выпусков — Windows 98 и Windows Millennium Edition). Вот список некоторых архитектурных различий и преимуществ Windows NT (и ее последующих выпусков) над Windows 95 (и ее последующими выпусками).

- Windows NT поддерживает многопроцессорные системы, а Windows 95 — нет.
- Файловая система Windows NT поддерживает средства защиты, например управление избирательным доступом (discretionary access control). В файловой системе Windows 95 этого нет.
- Windows NT — полностью 32-разрядная (а теперь и 64-разрядная) операционная система, в ней нет 16-разрядного кода, кроме того, который предназначен для выполнения 16-разрядных Windows-приложений. Windows 95 содержит большой объем старого 16-разрядного кода из предшествующих операционных систем — Windows 3.1 и MS-DOS.
- Windows NT полностью реентерабельна, а многие части Windows 95 нереентерабельны (в основном это касается 16-разрядного кода, взятого из Windows 3.1). Большинство функций, связанных с графикой и управлением окнами (GDI и USER), включают именно нереентерабельный код. Когда 32-разрядное приложение в Windows 95 пытается вызвать системный сервис, реализованный как нереентерабельный 16-разрядный код, оно должно сначала получить общесистемную блокировку (или мьютекс), чтобы предотвратить вход других потоков в нереентерабельную кодовую базу. Еще хуже, что 16-разрядное приложение удерживает такую блокировку в течение всего времени своего выполнения. В итоге, хотя ядро Windows 95 содержит 32-разрядный планировщик с поддержкой многопоточности и вытесняющей многозадачности, приложения часто работают как однопоточные из-за того, что большая часть системы реализована как нереентерабельный код.
- Windows NT позволяет выполнять 16-разрядные Windows-приложения в выделенном адресном пространстве, а Windows 95 всегда выполняет такие приложения в общем адресном пространстве, в котором они могут навредить друг другу и привести к зависанию системы.
- Разделяемая (общая) память процесса в Windows NT видна только тем процессам, которые имеют проекцию на один и тот же блок разделяемой памяти. В Windows 95 вся общая память видна и доступна для записи всем процессам. Таким образом, любой процесс

может что-то записать и повредить какие-то данные в общей памяти, используемые другими процессами.

- Некоторые критически важные страницы памяти, занимаемые операционной системой Windows 95, доступны для записи из пользовательского режима, а значит, обычное приложение может повредить содержимое этих страниц и привести к краху системы.

Единственное, что умеет Windows 95 и чего никогда не смогут делать операционные системы на основе Windows NT, — выполнять *все* старые программы для MS-DOS и Windows 3.1 (а именно программы, требующие прямого доступа к оборудованию), а также 16-разрядные драйверы устройств MS-DOS. Если одной из основных целей разработки Windows 95 была 100%-я совместимость с MS-DOS и Windows 3.1, то исходной целью разработки Windows NT — возможность выполнения *большинства* существующих 16-разрядных приложений при условии сохранения целостности и надежности системы.

Базовые концепции и термины

В книге будут часто встречаться ссылки на концепции и структуры, с которыми некоторые читатели, возможно, не знакомы. Здесь мы определимся с используемой в дальнейшем терминологией.

Windows API

Это системный интерфейс программирования в семействе операционных систем Microsoft Windows, включая Windows 2000, Windows XP, Windows Server 2003, Windows 95, Windows 98, Windows Millennium Edition (Me) и Windows CE. Каждая операционная система реализует разное подмножество Windows API. Windows 95, Windows 98, Windows Me и Windows CE в этой книге не рассматриваются.

ПРИМЕЧАНИЕ Windows API описывается в документации Platform Software Development Kit (SDK). (См. раздел «Platform Software Development Kit (SDK)» далее в этой главе.) Эту документацию можно бесплатно просмотреть на сайте *msdn.microsoft.com*. Она также поставляется с Microsoft Developer Network (MSDN) всех уровней подписки. (MSDN — это программа Microsoft для поддержки разработчиков. Подробности см. на сайте *msdn.microsoft.com*.) Отличное описание того, как программировать с использованием базового Windows API, см. в четвертом издании книги Джеффри Рихтера (Jeffrey Richter) «Microsoft Windows для профессионалов» (Русская Редакция, 2000).

До появления 64-разрядных версий Windows XP и Windows Server 2003 интерфейс программирования 32-разрядных версий операционных систем Windows назывался Win32 API, чтобы отличать его от исходного 16-разрядного Windows API. В этой книге термин «Windows API» относится к 32-разряд-

ному интерфейсу программирования Windows 2000, а также к 32- и 64-разрядным интерфейсам программирования Windows XP и Windows Server 2003.

Windows API включает тысячи вызываемых функций, которые сгруппированы в следующие основные категории:

- базовые сервисы (Base Services);
- сервисы компонентов (Component Services);
- сервисы пользовательского интерфейса (User Interface Services);
- сервисы графики и мультимедиа (Graphics and Multimedia Services);
- коммуникационное взаимодействие и совместная работа (Messaging and Collaboration);
- сети (Networking);
- Web-сервисы (Web Services).

Основное внимание в нашей книге уделяется внутреннему устройству ключевых базовых сервисов, в частности поддержки процессов и потоков (threads), управления памятью, ввода-вывода и защиты.

Как насчет .NET и WinFX?

.NET Framework состоит из библиотеки классов, называемой Framework Class Library (FCL), и общеязыковой исполняющей среды (Common Language Runtime, CLR), которая предоставляет среду для выполнения управляемого кода с такими возможностями, как компиляция по требованию (just-in-time compilation, JIT compilation), верификация типов, сбор мусора и защита по правам доступа кода (code access security). Благодаря этому CLR создает среду разработки, которая повышает продуктивность труда программистов и уменьшает вероятность появления распространенных ошибок программирования. Отличное описание .NET Framework и ее базовой архитектуры см. в книге Джеффри Рихтера «Программирование на платформе Microsoft .NET Framework» (Русская Редакция, 2003).

CLR реализована как классический COM-сервер, код которой хранится в стандартной Windows DLL пользовательского режима. Фактически все компоненты .NET Framework реализованы как стандартные Windows DLL пользовательского режима, занимающие уровень поверх неуправляемых функций Windows API. (Никакие компоненты .NET Framework не работают в режиме ядра.) На рис. 1-1 показаны взаимосвязи этих компонентов.

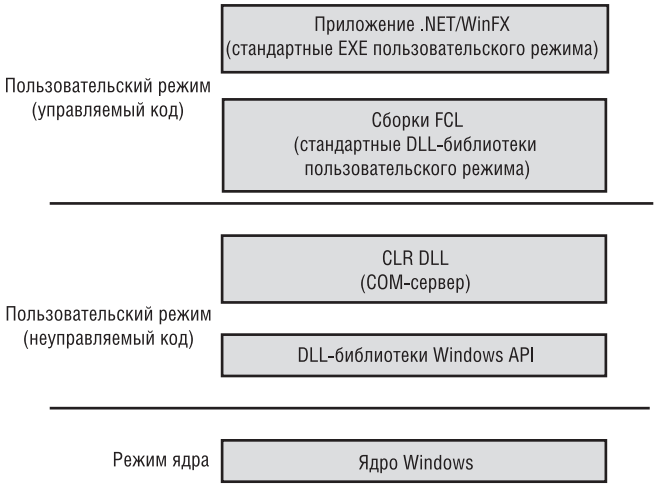


Рис. 1-1. *Взаимосвязи компонентов .NET Framework*

WinFX — «новый Windows API». Это результат эволюционного развития .NET Framework, которая будет поставляться с версией Windows под кодовым названием «Longhorn», следующим выпуском Windows. WinFX также можно установить в Windows XP и Windows Server 2003. WinFX образует фундамент для приложений следующего поколения, создаваемых для операционной системы Windows.

История создания Win32 API

Интересно, что поначалу Win32 не рассматривался как интерфейс программирования для Microsoft Windows NT. Поскольку проект Windows NT начинался как замена OS/2 версии 2, основным интерфейсом программирования был 32-разрядный OS/2 Presentation Manager API. Однако год спустя на рынке появилась Microsoft Windows 3.0, быстро ставшая очень популярной. В результате Microsoft сменила курс и перенацелила проект Windows NT на будущую замену семейства продуктов Windows, а не OS/2. Вот на этом-то перепутье и встал вопрос о создании Windows API — до этого Windows API существовал только как 16-разрядный интерфейс.

Хотя в Windows API должно было появиться много новых функций, отсутствующих в Windows 3.1, Microsoft решила сделать новый API по возможности совместимым с именами функций, семантикой и типами данных в 16-разрядном Windows API, чтобы максимально облегчить бремя переноса существующих 16-разрядных Windows-приложений в Windows NT. Поэтому тот, кто, впервые глядя на Windows API, удивляется, почему многие имена и интерфейсы функций кажутся противоречивыми, должен учитывать, что одной из причин такой противоречивости было стремление сделать Windows API совместимым со старым 16-разрядным Windows API.

Сервисы, функции и процедуры

Несколько терминов в документации Windows для пользователей и программистов имеет разный смысл в разных контекстах. Например, понятие «сервис» (service) может относиться к вызываемой функции операционной системы, драйверу устройства или серверному процессу (в последнем случае сервис часто называют службой). Ниже показано, что означают подобные термины в этой книге.

- **Функции Windows API** Документированные, вызываемые подпрограммы в Windows API, например *CreateProcess*, *CreateFile* и *GetMessage*.
- **Неуправляемые («родные») системные сервисы (или исполняемые системные сервисы)** Недокументированные низкоуровневые сервисы операционной системы, которые можно вызывать в пользовательском режиме. Так, *NtCreateProcess* — это внутрисистемный сервис, вызываемый Windows-функцией *CreateProcess* при создании нового процесса. (Определение неуправляемых функций см. в разделе «Диспетчеризация системных сервисов» главы 3.)
- **Функции (или процедуры) ядра** Подпрограммы внутри операционной системы Windows, которые можно вызывать только в режиме ядра (определение мы дадим чуть позже). Например, *ExAllocatePool* — процедура, вызываемая драйверами устройств для выделения памяти из системных куч (динамически распределяемых областей памяти) Windows.
- **Windows-сервисы** Процессы, запускаемые диспетчером управления сервисами в Windows. (Хотя в документации на реестр драйверы устройств Windows определяются как сервисы, мы не пользуемся таким термином в этой книге.) Например, сервис Task Scheduler выполняется в процессе пользовательского режима, который поддерживает команду *at* (аналогичную UNIX-команде *at* или *cron*).
- **DLL (динамически подключаемая библиотека)** Набор вызываемых подпрограмм, включенных в один двоичный файл, который приложения, использующие эти подпрограммы, могут динамически загружать во время своего выполнения. В качестве примера можно привести модули Msvcrt.dll (библиотека исполняющей подсистемы C) и Kernel32.dll (одна из библиотек подсистемы Windows API). DLL активно используются компонентами и приложениями Windows пользовательского режима. Преимущество DLL над статическими библиотеками в том, что приложения могут разделять DLL-модули, а Windows гарантирует, что в памяти будет находиться лишь по одному экземпляру используемых DLL.

Процессы, потоки и задания

Хотя на первый взгляд кажется, что *программа* и *процесс* — понятия практически одинаковые, они фундаментально отличаются друг от друга. *Программа* представляет собой статический набор команд, а *процесс* — это контейнер для набора ресурсов, используемых при выполнении экземпляра

программы. На самом высоком уровне абстракции процесс в Windows включает следующее:

- закрытое *виртуальное адресное пространство* — диапазон адресов виртуальной памяти, которым может пользоваться процесс;
- исполняемую программу — начальный код и данные, проецируемые на виртуальное адресное пространство процесса;
- список открытых описателей (handles) различных системных ресурсов — семафоров, коммуникационных портов, файлов и других объектов, доступных всем потокам в данном процессе;
- контекст защиты (security context), называемый *маркером доступа* (access token) и идентифицирующий пользователя, группы безопасности и привилегии, сопоставленные с процессом;
- уникальный идентификатор процесса (во внутрисистемной терминологии называемый идентификатором клиента);
- минимум один поток.

Каждый процесс также указывает на свой родительский процесс (процесс-создатель). Однако, если родитель существует, эта информация не обновляется. Поэтому есть вероятность, что некий процесс указывает на уже несуществующего родителя. Это не создает никакой проблемы, поскольку никто не полагается на наличие такой информации. Следующий эксперимент иллюстрирует данный случай.

ЭКСПЕРИМЕНТ: просмотр дерева процессов

Большинство утилит не отображает такой уникальный атрибут, как идентификатор родительского процесса. Значение этого атрибута можно получить программно или с помощью оснастки Performance, запросив значение счетчика Creating Process ID [Код (ID) создавшего процесса]. Дерево процессов показывается утилитой Tlist.exe (из Windows Debugging Tools), если вы указываете ключ /t. Вот образец вывода этой команды:

```
C:\>tlist /t
System Process (0)
System (2)
  smss.exe (21)
    csrss.exe (24)
    winlogon.exe (35)
      services.exe (41)
        spoolss.exe (69)
        llssrv.exe (94)
        LOCATOR.EXE (96)
        RpcSs.exe (112)
        inetinfo.exe (128)
      lsass.exe (44)
   nddeagnt.exe (119)
```

см. след. стр.

```
explorer.exe (123) Program Manager
  OSA.EXE (121)
  WINWORD.EXE (117) Microsoft Word - msch02(s).doc
  cmd.exe (72) Command Prompt - tlist /t
    tlist.EXE (100)
```

Взаимоотношения процессов (дочерний-родительский) Tlist показывает отступами. Имена процессов, родительские процессы которых на данный момент завершились, выравниваются по левому краю, потому что установить их родственные связи невозможно — даже если процессы-прапредки еще существуют. Windows сохраняет идентификатор только родительского процесса, так что проследить его создателя нельзя. Чтобы убедиться в этом, выполните следующие операции.

1. Откройте окно командной строки.
2. Наберите **start cmd** для запуска второго окна командной строки.
3. Откройте диспетчер задач.
4. Переключитесь на второе окно командной строки.
5. Введите **mspaint** для запуска Microsoft Paint.
6. Щелкните второе окно командной строки.
7. Введите **exit**. (Заметьте, что окно Paint остается.)
8. Переключитесь в диспетчер задач.
9. Откройте его вкладку Applications (Приложения).
10. Щелкните правой кнопкой мыши задачу Command Prompt (Командная строка) и выберите Go To Process (Перейти к процессам).
11. Щелкните процесс Cmd.exe, выделенный серым цветом.
12. Щелкнув правой кнопкой мыши, выберите команду End Process Tree (Завершить дерево процессов).
13. В окне Task Manager Warning (Предупреждение диспетчера задач) щелкните кнопку Yes (Да).

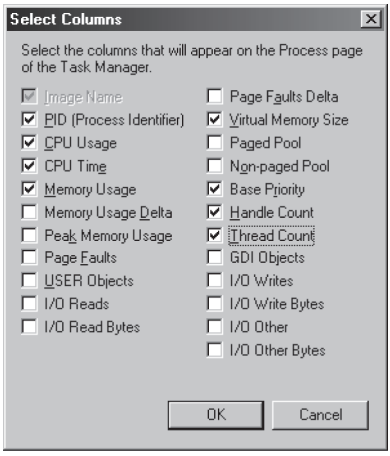
Первое окно командной строки исчезнет, но вы по-прежнему сможете наблюдать окно Paint, так как оно является внуком первого из завершенных процессов Command Prompt. А поскольку второй (родительский процесс Paint) тоже завершен, связь между родителем и внуком потеряна.

Для просмотра (и модификации) процессов и информации, связанной с ними, существует целый набор утилит. Следующие эксперименты демонстрируют, как получить ту или иную информацию о процессе с помощью некоторых из этих утилит. Они включаются непосредственно в саму Windows, а также в Windows Support Tools, Windows Debugging Tools, ресурсы Windows и Platform SDK; ряд утилит можно получить с сайта *www.sysinternals.com*. Многие из этих утилит выводят перекрывающиеся подмножества информации о базовых процессах и потоках, иногда идентифицируемые по разным именам.

Вероятно, наиболее широко применяемая утилита для анализа активности процессов — Task Manager (Диспетчер задач). (Любопытно, что в ядре Windows нет такого понятия, как задача, так что Task Manager на самом деле является инструментом для управления процессами.) Следующий эксперимент показывает разницу между тем, что Task Manager перечисляет как приложения и процессы.

ЭКСПЕРИМЕНТ: просмотр информации о процессах через диспетчер задач

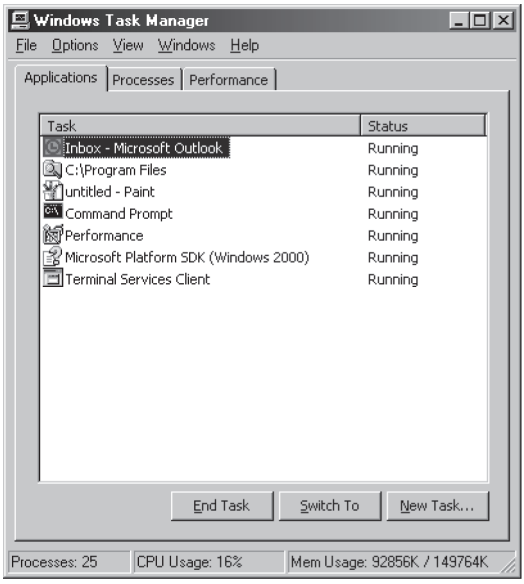
Диспетчер задач Windows отображает список выполняемых в системе процессов. Его можно запустить тремя способами: 1) нажав клавиши Ctrl+Shift+Esc; 2) щелкнув панель задач правой кнопкой мыши и выбрав команду Task Manager (Диспетчер задач); 3) нажав клавиши Ctrl+Alt+Del. После запуска диспетчера задач откройте вкладку Processes (Процессы). Заметьте, что процессы идентифицируются по имени образа, экземплярами которого они являются. В отличие от некоторых объектов в Windows процессам нельзя присваивать глобальные имена. Для просмотра более подробных сведений выберите из меню View (Вид) команду Select Columns (Выбрать столбцы) и укажите, какая дополнительная информация вас интересует.



Если вкладка Processes окна диспетчера задач со всей очевидностью показывает список процессов, то содержимое вкладки Applications (Приложения) нуждается в пояснениях. На ней отображается список видимых окон верхнего уровня всех объектов «рабочий стол» интерактивного объекта WindowStation. (По умолчанию существуют два объекта «рабочий стол», но вы можете создать дополнительные рабочие столы через Windows-функцию *CreateDesktop*.) Колонка Status (Состояние) дает представление о том, находится ли поток — владелец окна в состоянии ожидания Windows-сообщения. «Running» («Выполняется») означает, что поток ожидает ввода в окно, а «Not Responding» («Не

см. след. стр.

отвечает») — что не ожидает (т. е. занят либо ждет завершения операции ввода-вывода или освобождения какого-либо синхронизирующего объекта).



Вкладка Applications позволяет идентифицировать процесс, которому принадлежит поток, владеющий каким-либо окном задачи. Для этого щелкните правой кнопкой мыши имя задачи и выберите команду Go To Process (Перейти к процессам).

Утилита Process Explorer (с сайта *www.sysinternals.com*) показывает больше информации о процессах и потоках, чем любой другой доступный инструмент; вот почему она используется нами во многих экспериментах, которые вы увидите в этой книге. Ниже перечислены некоторые уникальные сведения, выводимые утилитой Process Explorer, и ее возможности:

- полное имя (вместе с путем) выполняемого образа;
- маркер защиты процесса (список групп и привилегий);
- выделение изменений в списке процессов и потоков;
- список сервисов внутри процессов — хостов сервисов с выводом отображаемого имени (display name) и описания;
- процессы, которые являются частью задания, и детальные сведения о заданиях;
- процессы, выполняющие .NET/WinFX-приложения, и сведения, специфичные для .NET (например, список доменов приложений и счетчики производительности, относящиеся к CLR);
- время запуска процессов и потоков;

- полный список файлов, проецируемых в память (не только DLL-модулей);
- возможность приостановки процесса;
- возможность принудительного завершения индивидуальных потоков;
- простота выявления процессов, использующих наибольшую долю процессорного времени за определенный период. (Оснастка Performance позволяет просматривать процент использования процессора для заданного набора процессов, но не показывает автоматически процессы, созданные после начала сеанса мониторинга.)

Process Explorer также упрощает доступ к информации, предоставляемой другими утилитами, создавая единую точку ее просмотра:

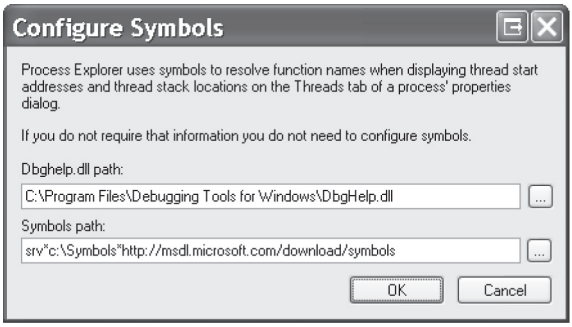
- дерево процессов с возможностью свертывания отдельных частей этого дерева;
- открытые описатели в процессе без предварительной настройки (утилиты Microsoft для вывода открытых описателей требуют предварительной установки общесистемного флага и перезагрузки);
- список DLL (и файлов, проецируемых в память) в каком-либо процессе;
- активность потоков в каком-либо процессе;
- стеки потоков пользовательского режима (с сопоставлением адресов именам, используя механизм поддержки символов для инструментов отладки);
- стеки системных потоков режима ядра (с сопоставлением адресов именам, используя механизм поддержки символов для инструментов отладки);
- разница в переключении контекстов (context switch delta) (более наглядное представление активности процессора, как поясняется в главе 6);
- лимиты памяти режима ядра (пулов подкачиваемой и неподкачиваемой памяти) (остальные утилиты показывают только текущие размеры).

Попробуем провести первый эксперимент с помощью Process Explorer.

ЭКСПЕРИМЕНТ: просмотр детальных сведений о процессах с помощью Process Explorer

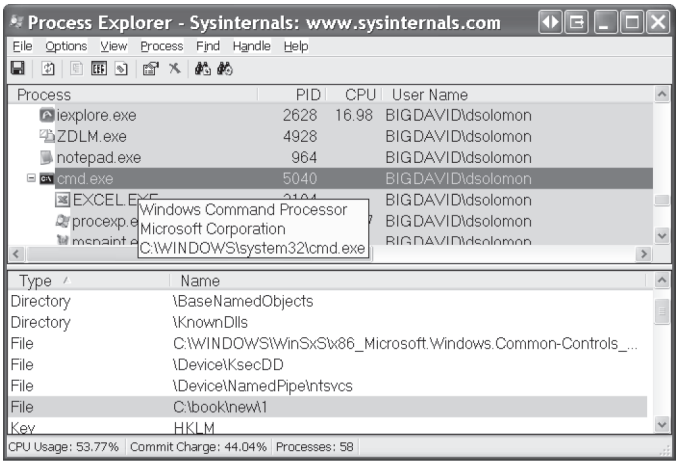
Скачайте последнюю версию Process Explorer с сайта *www.sysinternals.com* и запустите ее. При первом запуске вы увидите сообщение о том, что на данный момент символы не сконфигурированы. Когда они корректно сконфигурированы, Process Explorer может обращаться к символьной информации для отображения символьного имени стартовой функции потока и функций в его стеке вызовов (для этого нужно дважды щелкнуть процесс и выбрать вкладку Threads). Эта информация полезна для идентификации того, что именно делают потоки внутри процесса. Для доступа к символам вы должны установить Debugging Tools (об этом мы еще поговорим в данной главе). Потом щелкнуть Options, выбрать Configure Symbols и набрать подходящий путь Symbols. Например:

см. след. стр.



В предыдущем примере для доступа к символам использовался сервер символов по требованию (on-demand symbol server), а копии файлов символов хранились на локальном компьютере в папке c:\symbols. Подробнее о конфигурировании сервера символов см. по ссылке *http://www.microsoft.com/whdc/ddk/debugging/symbols.mspx*.

При запуске Process Explorer по умолчанию выводит список процессов в верхней половине окна, а список открытых описателей для выбранного на данный момент процесса — в нижней половине. Если вы задержите курсор мыши над именем процесса, Process Explorer также показывает описание образа, название компании и полный путь.



Вот как использовать некоторые базовые возможности Process Explorer:

1. Отключите нижнюю секцию, сбросив View, Show Lower Pane. (Нижняя секция может отображать открытые описатели или проецируемые DLL и файлы — об этом речь пойдет в главах 3 и 7.)
2. Обратите внимание на то, что процессы, являющиеся хостами сервисов, по умолчанию выделяются розовым цветом. Ваши собственные процессы выделяются синим. (Эти цвета можно настроить.)

3. Задержите курсор мыши над именем образа и обратите внимание на то, что в подсказке отображается полный путь.
4. Щелкните View, Select Columns и добавьте путь образа.
5. Отсортируйте по колонке процессов и вы увидите, что представление в виде дерева исчезло. (Вы можете либо вывести представление в виде дерева, либо сортировать по любой из отображаемых колонок.) Снова щелкните для сортировки по алфавиту в обратном порядке (от Z к A). После этого очередной щелчок вернет представление в виде дерева.
6. Сбросьте View, Show Processes From All Users для отображения только ваших процессов.
7. Перейдите в Options, Difference Highlight Duration и смените значение на 5 секунд. Потом запустите новый процесс (какой угодно) и обратите внимание на то, что этот процесс выделяется зеленым в течение 5 секунд. Закройте новый процесс и заметьте, что этот процесс выделяется красным в течение 5 секунд, прежде чем исчезнуть из древовидного списка. Эта функция может пригодиться для обнаружения создаваемых и завершаемых процессов в системе.
8. Наконец, дважды щелкните какой-нибудь процесс и изучите вкладки, доступные в окне свойств процесса. (Эти вкладки понадобятся нам в дальнейших экспериментах; там же мы поясним, какую информацию они сообщают.)

Поток (thread) — некая сущность внутри процесса, получающая процессорное время для выполнения. Без потока программа процесса не может выполняться. Поток включает следующие наиболее важные элементы:

- содержимое набора регистров процессора, отражающих состояние процессора;
- два стека, один из которых используется потоком при выполнении в режиме ядра, а другой — в пользовательском режиме;
- закрытую область памяти, называемую локальной памятью потока (thread-local storage, TLS) и используемую подсистемами, библиотеками исполняющих систем (run-time libraries) и DLL;
- уникальный идентификатор потока (во внутрисистемной терминологии также называемый идентификатором клиента: идентификаторы процессов и потоков генерируются из одного пространства имен и никогда не перекрываются);
- иногда потоки обладают своим контекстом защиты, который обычно используется многопоточными серверными приложениями, подменяющими контекст защиты обслуживаемых клиентов.

Переменные регистры, стеки и локальные области памяти называются *контекстом потока*. Поскольку эта информация различна на каждой аппаратной платформе, на которой может работать Windows, соответствующая

структура данных специфична для конкретной платформы. Windows-функция *GetThreadContext* предоставляет доступ к этой аппаратно-зависимой информации (называемой блоком CONTEXT).

Волокна и потоки

Волокна (fibers) позволяют приложениям планировать собственные «потоки» выполнения, не используя встроенный механизм планирования потоков на основе приоритетов. Волокна часто называют «облегченными» потоками. Они невидимы ядру, так как Kernel32.dll реализует их в пользовательском режиме. Для использования волокна нужно вызвать Windows-функцию *ConvertThreadToFiber*, которая преобразует поток в волокно. Полученное волокно может создавать дополнительные волокна через функцию *CreateFiber* (у каждого волокна может быть свой набор волокон). Выполнение волокна (в отличие от потока) не начинается до тех пор, пока оно не будет вручную выбрано вызовом *SwitchToFiber*. Волокно работает до завершения или до переключения процессора на другое волокно вызовом все той же *SwitchToFiber*. Подробнее о функциях, связанных с волокнами, см. документацию Platform SDK.

Хотя у потоков свой контекст выполнения, каждый поток внутри одного процесса делит его виртуальное адресное пространство (а также остальные ресурсы, принадлежащие процессу). Это означает, что все потоки в процессе могут записывать и считывать содержимое памяти любого из потоков данного процесса. Однако потоки не могут случайно сослаться на адресное пространство другого процесса. Исключение возможно в ситуации, когда тот предоставляет часть своего адресного пространства как *раздел общей памяти* (shared memory section), в Windows API называемый объектом «проекция файла» (file mapping object), или когда один из процессов имеет право на открытие другого процесса и использует функции доступа к памяти между процессами, например *ReadProcessMemory* и *WriteProcessMemory*.

Кроме закрытого адресного пространства и одного или нескольких потоков у каждого процесса имеются идентификация защиты и список открытых описателей таких объектов, как файлы и разделы общей памяти, или синхронизирующих объектов вроде мьютексов, событий и семафоров (рис. 1-2).

Каждый процесс обладает контекстом защиты, который хранится в объекте — *маркере доступа*. Маркер доступа содержит идентификацию защиты и определяет полномочия данного процесса. По умолчанию у потока нет собственного маркера доступа, но он может получить его, и это позволит ему подменять контекст защиты другого процесса (в том числе выполняемого на удаленной системе Windows). Подробнее на эту тему см. главу 8.

Дескрипторы виртуальных адресов (virtual address descriptors, VAD) — это структуры данных, используемые диспетчером памяти для учета виртуальных адресов, задействованных процессом (см. главу 7).

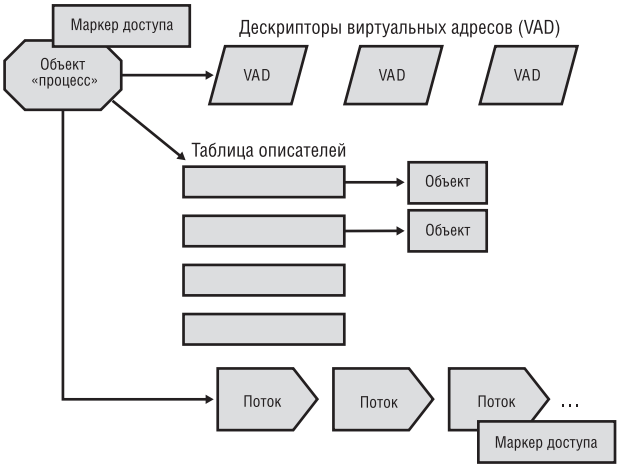


Рис. 1-2. *Процесс и его ресурсы*

Windows предоставляет расширение для модели процессов — *задания* (jobs). Они предназначены в основном для того, чтобы группами процессов можно было оперировать и управлять как единым целым. Объект-задание позволяет устанавливать определенные атрибуты и накладывать ограничения на процесс или процессы, сопоставленные с заданием. В этом объекте также хранится информация обо всех процессах, которые были сопоставлены с заданием, но к настоящему времени уже завершены. В каких-то отношениях объект-задание компенсирует отсутствие иерархического дерева процессов в Windows, а в каких-то — даже превосходит по своим возможностям дерево процессов UNIX.

Более детальное описание внутренней структуры заданий, процессов и потоков, механизмов создания потоков и процессов, а также алгоритмов планирования потоков вы найдете в главе 6.

Виртуальная память

В Windows реализована система виртуальной памяти, основанная на плоском (линейном) адресном пространстве. Она создает каждому процессу иллюзию того, что у него есть собственное большое и закрытое адресное пространство. Виртуальная память дает логическое представление, не обязательно соответствующее структуре физической памяти. В период выполнения диспетчер памяти, используя аппаратную поддержку, транслирует, или *проецирует* (maps), виртуальные адреса на физические, по которым реально хранятся данные. Управляя проецированием и защитой страниц памяти, операционная система гарантирует, что ни один процесс не помешает другому и не сможет повредить данные самой операционной системы. На рис. 1-3 показано, как три смежные страницы виртуальной памяти проецируются на три разрозненные страницы физической памяти.

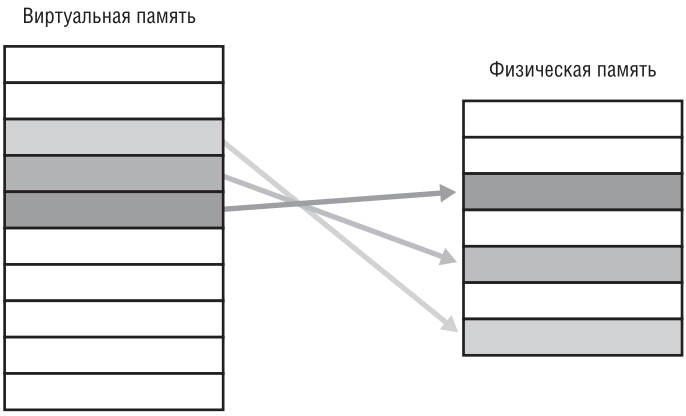


Рис. 1-3. *Проецирование виртуальной памяти на физическую*

Поскольку у большинства компьютеров объем физической памяти намного меньше общего объема виртуальной памяти, задействованной выполняемыми процессами, диспетчер памяти перемещает, или подкачивает (pages), часть содержимого памяти на диск. Подкачка данных на диск освобождает физическую память для других процессов или самой операционной системы. Когда поток обращается к странице виртуальной памяти, сброшенной на диск, диспетчер виртуальной памяти загружает эту информацию с диска обратно в память. Для использования преимуществ подкачки в приложениях никакого дополнительного кода не требуется, так как диспетчер памяти опирается на аппаратную поддержку этого механизма.

Размер виртуального адресного пространства зависит от конкретной аппаратной платформы. На 32-разрядных x86-системах теоретический максимум для общего виртуального адресного пространства составляет 4 Гб. По умолчанию Windows выделяет нижнюю половину этого пространства (в диапазоне адресов от x00000000 до x7FFFFFFF) процессам, а вторую половину (в диапазоне адресов от x80000000 до xFFFFFFFF) использует в собственных целях. Windows 2000 Advanced Server, Windows 2000 Datacenter Server, Windows XP (SP2 и выше) и Windows Server 2003 поддерживают загрузочные параметры /3GB и /USERVA, которые указываются в файле Boot.ini (см. главу 5), что позволяет процессам, выполняющим программы со специальным флагом в заголовке исполняемого образа, использовать до 3 Гб закрытого адресного пространства и оставляет операционной системе только 1 Гб. Этот вариант дает возможность приложению вроде сервера базы данных хранить в адресном пространстве своего процесса бóльшие порции базы данных и тем самым уменьшить частоту проецирования отдельных представлений этой базы. Две структуры виртуальных адресных пространств, поддерживаемые 32-разрядной Windows, показаны на рис. 1-4.

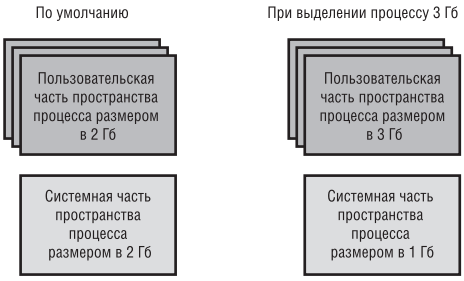


Рис. 1-4. *Структуры адресных пространств для 32-разрядных Windows*

Хотя три гигабайта лучше двух, этого все равно недостаточно для проецирования очень больших баз данных. В связи с этим в 32-разрядных Windows появился механизм Address Windowing Extension (AWE), который позволяет 32-разрядному приложению выделять до 64 Гб физической памяти, а затем проецировать представления (views), или окна (windows), на свое 2-гигабайтное виртуальное адресное пространство. Применение AWE усложняет управление проекциями виртуальной памяти на физическую, но снимает проблему прямого доступа к объему физической памяти, превышающему лимиты 32-разрядного адресного пространства процесса.

64-разрядная Windows предоставляет процессам гораздо большее адресное пространство: 7152 Гб на Itanium-системах и 8192 Гб на x64-системах. На рис. 1-5 показана упрощенная схема структур 64-разрядных адресных пространств (детали см. в главе 7). Заметьте, что эти размеры отражают не архитектурные лимиты для данных платформ, а ограничения реализации в текущих версиях 64-разрядной Windows.

Рис. 1-5. *Структуры адресных пространств для 64-разрядной Windows*

Подробнее о реализации диспетчера памяти, в том числе о трансляции адресов и управлении физической памятью в Windows, см. главу 7.

Режим ядра и пользовательский режим

Для предотвращения доступа приложений к критически важным данным операционной системы и устранения риска их модификации Windows использует два режима доступа к процессору (даже если он поддерживает более двух режимов): пользовательский (user mode) и ядра (kernel mode). Код

приложений работает в пользовательском режиме, тогда как код операционной системы (например, системные сервисы и драйверы устройств) — в режиме ядра. В режиме ядра предоставляется доступ ко всей системной памяти и разрешается выполнять любые машинные команды процессора. Предоставляя операционной системе более высокий уровень привилегий, чем прикладным программам, процессор позволяет разработчикам операционных систем реализовать такие архитектуры, которые не дают возможности сбойным приложениям нарушать стабильность работы всей системы.

ПРИМЕЧАНИЕ В архитектуре процессора Intel x86 определено четыре уровня привилегий, или *колец* (rings), предназначенных для защиты кода и данных системы от случайной или умышленной перезаписи кодом с меньшим уровнем привилегий. Windows использует уровень привилегий 0 (или кольцо 0) для режима ядра и уровень привилегий 3 (или кольцо 3) для пользовательского режима. Почему Windows использует только два уровня? Дело в том, что на некоторых из ранее поддерживавшихся аппаратных платформ (например, Compaq Alpha и Silicon Graphics MIPS) реализовано лишь два уровня привилегий.

Хотя каждый Windows-процесс имеет свою (закрытую) память, код операционной системы и драйверы устройств, работающие в режиме ядра, делят единое виртуальное адресное пространство. Каждая страница в виртуальной памяти помечается тэгом, определяющим, в каком режиме должен работать процессор для чтения и/или записи данной страницы. Страницы в системном пространстве доступны лишь в режиме ядра, а все страницы в пользовательском адресном пространстве — в пользовательском режиме. Страницы только для чтения (например, содержащие лишь исполняемый код) ни в каком режиме для записи недоступны.

Windows не предусматривает никакой защиты системной памяти от компонентов, работающих в режиме ядра. Иначе говоря, код операционной системы и драйверов устройств в режиме ядра получает полный доступ к системной памяти и может обходить средства защиты Windows для обращения к любым объектам. Поскольку основная часть кода Windows выполняется в режиме ядра, крайне важно, чтобы компоненты, работающие в этом режиме, были тщательно продуманы и протестированы.

Это также подчеркивает, насколько надо быть осторожным при загрузке драйвера устройства от стороннего поставщика: перейдя в режим ядра, он получит полный доступ ко всем данным операционной системы. Такая уязвимость стала одной из причин, по которым в Windows введен механизм проверки цифровых подписей драйверов, предупреждающий пользователя о попытке установки неавторизованного (неподписанного) драйвера (подробнее на эту тему см. главу 9). Кроме того, механизм Driver Verifier (верификатор драйверов) помогает разработчикам драйверов устройств находить в них ошибки (вызывающие, например, утечку памяти или переполнения буферов). Driver Verifier поясняется в главе 7.

Как вы увидите в главе 2, прикладные программы могут переключаться из пользовательского режима в режим ядра, обращаясь к системному сервису. Например, Windows-функции *ReadFile* в ходе своего выполнения приходится вызывать внутреннюю подпрограмму Windows — она-то и считывает данные из файла. Так как эта подпрограмма обращается к внутрисистемным структурам данных, она должна выполняться в режиме ядра. Переключение из пользовательского режима в режим ядра осуществляется специальной командой процессора. Операционная система перехватывает эту команду, обнаруживает запрос системного сервиса, проверяет аргументы, которые поток передал системной функции, и выполняет внутреннюю подпрограмму. Перед возвратом управления пользовательскому потоку процессор переключается обратно в пользовательский режим. Благодаря этому операционная система защищает себя и свои данные от возможной модификации пользовательскими процессами.

ПРИМЕЧАНИЕ Переключение из пользовательского режима в режим ядра (и обратно) не влияет на планирование потока, так как контекст в этом случае не переключается. О диспетчеризации системных сервисов см. главу 3.

Так что ситуация, когда пользовательский поток часть своего времени работает в пользовательском режиме, а часть — в режиме ядра, совершенно нормальна. А поскольку подсистема, отвечающая за поддержку графики и окон, функционирует в режиме ядра, то приложения, интенсивно работающие с графикой, большую часть времени действуют в режиме ядра, а не в пользовательском режиме. Самый простой способ проверить это — запустить приложение вроде Microsoft Paint или Microsoft Pinball и с помощью одного из счетчиков оснастки Performance (Производительность), перечисленных в таблице 1-2, понаблюдать за показателями времени работы в пользовательском режиме и в режиме ядра.

Таблица 1-2. *Счетчики, позволяющие контролировать время работы в различных режимах*

Объект: счетчик	Описание
Processor: % Privileged Time (Процессор: % работы в привилегированном режиме)	Процентная доля времени, в течение которого отдельный процессор (или все процессоры) работал в режиме ядра
Processor: % User Time (Процессор: % работы в пользовательском режиме)	Процентная доля времени, в течение которого отдельный процессор (или все процессоры) работал в пользовательском режиме
Process: % Privileged Time (Процесс: % работы в привилегированном режиме)	Процентная доля времени, в течение которого потоки данного процесса выполнялись в режиме ядра
Process: % User Time (Процесс: % работы в пользовательском режиме)	Процентная доля времени, в течение которого потоки данного процесса выполнялись в пользовательском режиме

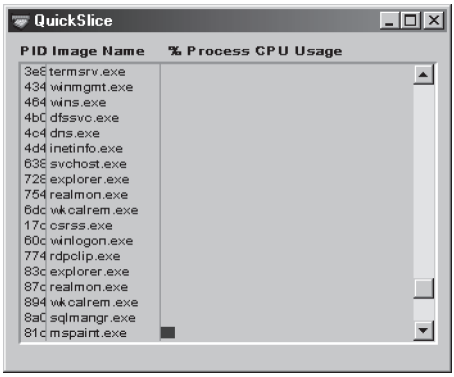
см. след. стр.

Таблица 1-2. (*продолжение*)

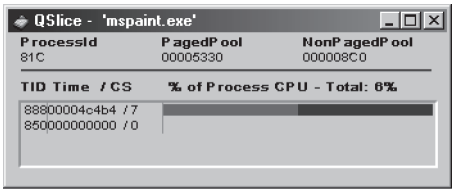
Объект: счетчик	Описание
Thread: % Privileged Time (Поток: % работы в привилегированном режиме)	Процентная доля времени, в течение которого данный поток выполнялся в режиме ядра
Thread: % User Time (Поток: % работы в пользовательском режиме)	Процентная доля времени, в течение которого данный поток выполнялся в пользовательском режиме

ЭКСПЕРИМЕНТ: наблюдение за активностью потоков с помощью QuickSlice

QuickSlice позволяет в динамике наблюдать за соотношением времени, проведенного каждым процессом в режиме ядра и в пользовательском режиме. На диаграмме красная часть столбца отражает количество процессорного времени в режиме ядра, а синяя — в пользовательском режиме. (Хотя в книге эти столбцы воспроизведены в черно-белом цвете, на самом деле они всегда красные и синие.) Сумма всех показателей, отображаемых столбцами в окне QuickSlice, должна соответствовать 100% процессорного времени. Для запуска QuickSlice щелкните кнопку Start (Пуск), выберите команду Run (Выполнить) и введите **Qslice.exe** (в переменной PATH должен быть указан путь к ресурсам Windows). Например, попробуйте запустить такое интенсивно использующее графику приложение, как Paint (Mspaint.exe). Откройте QuickSlice, расположив его окно рядом с окном Paint, и нарисуйте в Paint несколько кривых. В это время вы сможете наблюдать за выполнением Mspaint.exe в окне QuickSlice, как показано ниже.



Чтобы получить дополнительную информацию о потоках процесса, дважды щелкните имя нужного процесса или соответствующий цветной столбик на диаграмме. Вы увидите список потоков этого процесса и относительное процессорное время, используемое каждым потоком (в рамках процесса, а не всей системы).



ЭКСПЕРИМЕНТ: режим ядра и пользовательский режим

С помощью оснастки Performance вы можете выяснить, сколько времени ваша система работает в режиме ядра и в пользовательском режиме.

1. Запустите оснастку Performance (Производительность), открыв меню Start (Пуск) и последовательно выбрав команды Programs (Программы), Administrative Tools (Администрирование), Performance (Производительность).
2. Щелкните на панели инструментов кнопку Add (Добавить) (на этой кнопке изображен большой знак плюс).
3. Выберите в списке объект Processor (Процессор), щелкните счетчик % Privileged Time (% работы в привилегированном режиме) и, удерживая клавишу Ctrl в нажатом состоянии, щелкните счетчик % User Time (% работы в пользовательском режиме).
4. Щелкните кнопку Add (Добавить), а затем Close (Закрыть).
5. Быстро подвигайте мышью. При этом вы должны заметить всплеск на линии % Privileged Time (рис. 1-6), который отражает время, затраченное на обслуживание прерываний от мыши, и время, понадобившееся подсистеме поддержки окон на отрисовку графики (эта подсистема, как поясняется в главе 2, работает преимущественно как драйвер устройства в режиме ядра).
6. Закончив, щелкните на панели инструментов кнопку New Counter Set (Новый набор счетчиков) (или просто закройте оснастку).

За той же активностью можно понаблюдать через Task Manager (Диспетчер задач). Просто перейдите в нем на вкладку Performance (Быстродействие), а затем выберите из меню View (Вид) команду Show Kernel Times (Вывод времени ядра). Процент загруженности процессора отражается зеленым цветом, а процент времени работы в режиме ядра — красным.

см. след. стр.

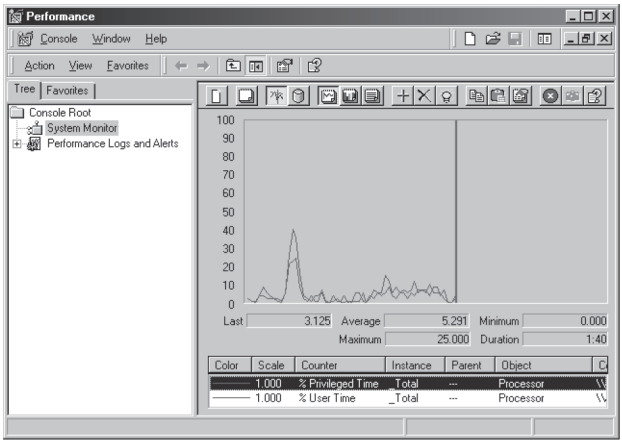


Рис. 1-6. *Оснастка Performance, показывающая, как распределяется время работы процессора между двумя режимами — пользовательским и ядра*

Чтобы увидеть, как сама оснастка Performance использует время в двух режимах, запустите ее снова, но добавьте те же счетчики для объекта Process (Процесс).

1. Если вы закрыли оснастку Performance, снова запустите ее. (Если она уже работает, откройте новый экран, щелкнув на панели инструментов кнопку New Counter Set.)
2. Щелкните кнопку Add на панели инструментов.
3. Выберите в списке объект Process.
4. Выберите счетчики % Privileged Time и % User Time.
5. В списке экземпляров объекта выберите все процессы (кроме процесса _Total).
6. Щелкните кнопку Add, а затем Close.
7. Быстро подвигайте мышью.
8. Нажмите комбинацию клавиш Ctrl+H для активизации режима выделения — текущий выбранный счетчик будет выделен белым цветом в Windows 2000 и черным в Windows XP или Windows Server 2003.
9. Прокрутите список всех счетчиков в нижней части окна оснастки, чтобы определить процессы, потоки которых выполнялись при перемещении мыши, и обратите внимание на то, в каком режиме они выполнялись — пользовательском или ядра.

Вы должны заметить, как значения счетчиков для процесса оснастки Performance — ищите mmc в колонке Instance (Экземпляр) — резко увеличиваются при перемещении мыши, поскольку код приложения выполняется в пользовательском режиме, а вызываемые им Windows-функции — в режиме ядра. Вы также заметите, что при перемещении мыши увеличивается активность работы в режиме ядра пото-

ка процесса csrss. Он представляет поток необработанного ввода (raw input thread) подсистемы Windows, принимающий ввод от клавиатуры и мыши и передающий его процессу, к которому он подключен. (Подробнее о системных потоках см. главу 2.) Наконец, процесс с именем Idle, потоки которого, как вы убедитесь, тратят почти 100% своего времени в режиме ядра, на самом деле не является процессом. Это лжепроцесс, используемый для учета тактов процессора в состоянии простоя. Таким образом, когда Windows нечего делать, она предается этому занятию в режиме ядра.

Terminal Services и несколько сеансов

Terminal Services (службы терминала) обеспечивают в Windows поддержку нескольких интерактивных сеансов пользователей на одной системе. С помощью Terminal Services удаленный пользователь может установить сеанс на другой машине, зарегистрироваться на ней и запускать приложения на сервере. Сервер предоставляет клиенту графический пользовательский интерфейс (GUI), а клиент возвращает серверу пользовательский ввод. (Это отличается от того, как ведет себя X Windows на UNIX-системах, где разрешается выполнять индивидуальные приложения на сервере, а клиенту предоставляется удаленный дисплей, так как удаленным является весь сеанс пользователя — не только одно приложение.)

Первый сеанс входа на физической консоли компьютера считается консольным сеансом, или нулевым сеансом (session zero). Дополнительные сеансы можно создать с помощью программы соединения с удаленным рабочим столом (Mstsc.exe), а в Windows XP — через механизм быстрого переключения пользователей (об этом позже).

Возможность создания удаленного сеанса поддерживается Windows 2000 Server, но не Windows 2000 Professional. Windows XP Professional позволяет одному удаленному пользователю подключаться к машине, однако если кто-то начинает процедуру входа в консоли, рабочая станция блокируется (т. е. систему можно использовать либо локально, либо удаленно, но не и то, и другое одновременно).

Windows 2000 Server и Windows Server 2003 поддерживают два одновременных удаленных сеанса. (Это упрощает удаленное управление, например облегчает применение инструментов, требующих от администратора входа на удаленный компьютер.) Windows 2000 Advanced Server, Datacenter Server и все издания Windows Server 2003 способны поддерживать более двух сеансов одновременно при условии правильного лицензирования и настройки системы в качестве сервера терминала.

Хотя Windows XP Home и Professional не поддерживают несколько удаленных подключений к рабочему столу, они все же поддерживают несколько сеансов, созданных локально через механизм быстрого переключения пользователей. (Этот механизм отключается в Windows XP Professional, если система присоединяется к домену.) Когда пользователь выбирает выключение

своего сеанса вместо выхода [например, последовательным выбором Start (Пуск), Log Off (Выход из системы) и Switch User (Смена пользователя) или нажатием клавиши L при одновременном удерживании клавиши Windows], текущий сеанс (т. е. процессы, выполняемые в этом сеансе, и все структуры данных, глобальные для сеанса и описывающие его) остается в системе, а Windows возвращается к основному окну входа. Если в систему входит новый пользователь, создается новый сеанс.

Для приложений, которым нужно знать, выполняются ли они в сеансе сервера терминала, предназначен набор Windows API-функций, позволяющих программно распознавать такую ситуацию и контролировать различные аспекты служб терминала. (Детали см. в Platform SDK.)

В главе 2 кратко описывается, как создаются сеансы, и проводится несколько экспериментов, показывающих, как просматривать информацию о сеансе с помощью различных инструментов, включая отладчик ядра. В разделе «Диспетчер объектов» главы 3 поясняется, как создается сеансовый экземпляр системного пространства имен для объектов и как приложения могут узнавать о других своих экземплярах в той же системе. Наконец, в главе 7 рассказывается, как диспетчер памяти настраивает данные, глобальные для сеанса, и управляет ими.

Объекты и описатели

В операционной системе Windows *объект* — это единственный экземпляр периода выполнения (run-time instance) статически определенного типа объекта. *Тип объекта* состоит из общесистемного типа данных, функций, оперирующих экземплярами этого типа данных, и набора атрибутов. Если вы пишете Windows-приложения, вам наверняка знакомы такие объекты, как процесс, поток, файл и событие, — продолжать можно еще долго. Эти объекты базируются на объектах более низкого уровня, создаваемых и управляемых Windows. В Windows процесс является экземпляром объекта типа «процесс», файл — экземпляром объекта типа «файл» и т. д.

Атрибут объекта (object attribute) — это поле данных в объекте, частично определяющее состояние этого объекта. Например, объект типа «процесс», имеет атрибуты, в число которых входят идентификатор процесса, базовый приоритет и указатель на объект маркера доступа. Методы объекта (средства для манипулирования объектами) обычно считывают или изменяют какие-либо атрибуты. Так, метод *open* процесса мог бы принимать идентификатор процесса и возвращать указатель на этот объект.

> **ПРИМЕЧАНИЕ** Не путайте параметр *ObjectAttributes*, предоставляемый вызывающей программой при создании объекта через Windows API или его родные сервисы, с термином «атрибуты объекта», имеющим более общий смысл.

Самое главное различие между объектом и обычной структурой данных заключается в том, что внутренняя структура объекта скрыта. Чтобы получить данные из объекта или записать в него какую-то информацию, вы долж-

ны вызвать его сервис. Прямое чтение или изменение данных внутри объекта невозможно. Тем самым реализация объекта отделяется от кода, который просто использует его, а это позволяет менять реализацию объекта, не модифицируя остальной код.

Объекты очень удобны для поддержки четырех важных функций операционной системы:

- присвоения понятных имен системным ресурсам;
- разделения ресурсов и данных между процессами;
- защиты ресурсов от несанкционированного доступа;
- учета ссылок (благодаря этому система узнает, когда объект больше не используется, и автоматически уничтожает его).

Не все структуры данных в Windows являются объектами. В объекты помещаются лишь те данные, которые нужно разделять, защищать, именовать или делать доступными программам пользовательского режима (через системные сервисы). Структуры, используемые только одним из компонентов операционной системы для поддержки каких-то внутренних функций, к объектам не относятся. Подробнее объекты и их описатели (ссылки на экземпляр объекта) рассматриваются в главе 3.

Безопасность

Windows с самого начала разрабатывалась как защищенная система, удовлетворяющая требованиям различных правительственных и промышленных стандартов безопасности, например спецификации Common Criteria for Information Technology Security Evaluation (CCITSE). Подтверждение правительством рейтинга безопасности операционной системы позволяет ей конкурировать в сферах, требующих повышенной защиты. Разумеется, многим из этих требований должна удовлетворять любая многопользовательская система.

Базовые возможности защиты в Windows таковы: избирательная защита любых разделяемых системных объектов (файлов, каталогов, процессов, потоков и т. д.), аудит безопасности (для учета пользователей и инициируемых ими операций), аутентификация паролей при входе и предотвращение доступа одного из пользователей к неинициализированным ресурсам (например, к памяти или дисковому пространству), освобожденным другим пользователем.

Windows поддерживает два вида контроля доступа к объектам. Первый из них — *управление избирательным доступом* (discretionary access control) — является механизмом, который как раз и связывается большинством пользователей с защитой. Это метод, при котором владельцы объектов (например, файлов или принтеров) разрешают или запрещают доступ к ним для других пользователей. При входе пользователь получает набор удостоверений защиты (security credentials), или контекст защиты (security context). Когда он пытается обратиться к объекту, его контекст защиты сверяется со списком управления доступом (access control list, ACL) для данного объекта, чтобы определить, имеет ли он разрешение на выполнение запрошенной операции.

Второй метод — *управление привилегированным доступом* (privileged access control) — необходим в тех случаях, когда управления избирательным доступом недостаточно. Данный метод гарантирует, что пользователь сможет обратиться к защищенным объектам, даже если их владелец недоступен. Например, если какой-то сотрудник увольняется из компании, администратору нужно получить доступ к файлам, которые могли быть доступны только бывшему сотруднику. В таких случаях Windows позволяет администратору стать владельцем этих файлов и при необходимости управлять правами доступа к ним.

Защита пронизывает весь интерфейс Windows API. Подсистема Windows реализует защиту на основе объектов точно так же, как и сама операционная система. При первой попытке доступа приложения к общему (разделяемому) объекту подсистема Windows проверяет, имеет ли это приложение соответствующие права. Если проверка завершается успешно, подсистема Windows разрешает приложению доступ.

Подсистема Windows реализует защиту для общих объектов, часть из которых построена на основе родных объектов Windows. К Windows-объектам относятся объекты рабочего стола, меню, окна, файлы, процессы, потоки и ряд синхронизирующих объектов.

Детальное описание защиты в Windows см. в главе 8.

Реестр

Если вы работали хоть с какой-нибудь операционной системой Windows, то, вероятно, слышали о реестре или даже просматривали его. Рассказать о внутреннем устройстве Windows, не упоминая реестр, вряд ли возможно, так как это системная база данных с информацией, необходимой для загрузки и конфигурирования системы; в ней содержатся общесистемные параметры, контролирующие работу Windows, база данных защиты и конфигурационные настройки, индивидуальные для каждого пользователя.

Кроме того, реестр — это окно, через которое можно заглянуть в переменные системные данные, чтобы, например, выяснить текущее состояние аппаратной части системы (какие драйверы устройств загружены, какие ресурсы они используют и т. д.) или значения счетчиков производительности Windows. Счетчики производительности, которые на самом деле в реестре не хранятся, доступны через функции реестра (см. главу 4).

Хотя у многих пользователей и администраторов Windows никогда не возникает необходимости работать непосредственно с реестром (бóльшую часть параметров можно просматривать или модифицировать с помощью стандартных административных утилит), он все же является источником полезной информации о внутренних структурах данных Windows, так как содержит множество параметров, влияющих на быстродействие и поведение системы. (Будьте крайне осторожны, напрямую изменяя параметры реестра: любые изменения могут отрицательно сказаться на быстродействии или, что гораздо хуже, привести к краху системы.)

Ссылки на различные разделы реестра, относящиеся к описываемым компонентам, будут встречаться на протяжении всей книги. Большинство таких

разделов находится в ветви HKEY_LOCAL_MACHINE, которую мы сокращенно называем HKLM. Подробнее о реестре и его внутренней структуре см. главу 4.

Unicode

Windows отличается от большинства других операционных систем тем, что в качестве внутреннего формата для хранения и обработки текстовых строк использует Unicode. Unicode — это стандартная кодировка, которая поддерживает многие известные в мире наборы символов и в которой каждый символ представляется 16-битным (двухбайтовым) кодом. (Подробнее о Unicode см. *www.unicode.org* и документацию на компакт-дисках MSDN Library.)

Поскольку многие приложения имеют дело с 8-битными (однобайтовыми) ANSI-символами, Windows-функции, принимающие строковые параметры, существуют в двух версиях: для Unicode и для ANSI. В Windows 95, Windows 98 и Windows ME реализована лишь часть Unicode-версий Windows-функций, поэтому приложения, рассчитанные на выполнение как в одной из этих операционных систем, так и в NT-подобных Windows, обычно используют ANSI-версии функций. Если вы вызываете ANSI-версию Windows-функции, входные строковые параметры перед обработкой системой преобразуются в Unicode, а выходные — из Unicode в ANSI (перед возвратом приложению). Таким образом при использовании в Windows устаревшего сервиса или фрагмента кода, написанного в расчете на ANSI-строки, эта операционная система будет вынуждена преобразовывать ANSI-символы в Unicode. Однако Windows никогда не преобразует данные внутри файлов — решения о том, в какой кодировке хранить текстовую информацию в файлах, принимают лишь сами приложения.

В предыдущих версиях Windows ее азиатский и ближневосточный выпуски представляли собой надмножество базовых американского и европейского выпусков, в которые включались дополнительные Windows-функции для обработки более сложных раскладок клавиатур и принципов ввода текста (например, набора текста справа налево). Начиная с Windows 2000, все языковые выпуски содержат одинаковые Windows-функции. Единая для всех стран двоичная кодовая база Windows способна поддерживать множество языков за счет простого добавления нужных компонентов языковой поддержки. Используя эти Windows-функции, разработчики могут создавать универсальные приложения, способные работать со множеством языков.

Изучение внутреннего устройства Windows

Хотя большая часть информации, представленная в этой книге, получена при чтении исходного кода Windows и общении с разработчиками, вы не обязаны принимать все на веру. Многие детали внутреннего устройства Windows можно вытащить на свет с помощью самых разнообразных средств, в том числе поставляемых с Windows, входящих в Windows Support Tools и ресурсы Windows, а также с использованием отладочных средств самой Windows. Чуть позже мы вкратце рассмотрим эти пакеты инструментальных средств.

Чтобы упростить вам исследование внутреннего устройства Windows, мы часто даем в книге врезки «Эксперимент» с пошаговыми инструкциями для изучения какого-либо аспекта поведения Windows. (Вы уже видели такие врезки в этой главе.) Советуем проводить эти эксперименты — это позволит увидеть в действии многие вещи, о которых рассказывается в книге.

В таблице 1-3 перечислены все используемые нами инструменты и утилиты.

Таблица 1-3. *Средства просмотра внутренней информации Windows*

Утилита	Имя образа	Источник
Startup Programs Viewer	AUTORUNS	*www.sysinternals.com*
Dependency Walker	DEPENDS	Support Tools, Platform SDK
DLL List	LISTDLLS	*www.sysinternals.com*
EFS Information Dumper	EFSDUMP	*www.sysinternals.com*
File Monitor	FILEMON	*www.sysinternals.com*
Global Flags	GFLAGS	Support Tools
Handle Viewer	HANDLE	*www.sysinternals.com*
Junction	JUNCTION	*www.sysinternals.com*
Отладчики ядра	WINDBG, KD	Средства отладки, Platform SDK, Windows DDK
Live Kernel Debugging	LIVEKD	Ресурсы Windows
Object Viewer	WINOBJ	*www.sysinternals.com*
Open Handles	OH	Ресурсы Windows
Page Fault Monitor	PFMON	Support Tools, Ресурсы Windows, Platform SDK
Pending File Moves	PENDMOVES	*www.sysinternals.com*
Performance	PERFMON.MSC	Утилита Windows
PipeList	PIPELIST	*www.sysinternals.com*
Pool Monitor	POOLMON	Support Tools, Windows DDK
Process Explorer	PROCEXP	*www.sysinternals.com*
Get SID	PSGETSID	*www.sysinternals.com*
Process Statistics	PSTAT	Support Tools, Ресурсы Windows, Platform SDK, *www.reskit.com*
Process Viewer	PVIEWER (в Support Tools) или PVIEW (в Platform SDK)	Support Tools, Platform SDK
Quick Slice	QSLICE	Ресурсы Windows
Registry Monitor	REGMON	*www.sysinternals.com*
Service Control	SC	Windows XP, Platform SDK, Ресурсы Windows
Task (Process) List	TLIST	Средства отладки
Task Manager	TASKMGR	Утилита Windows
TDImon	TDIMON	*www.sysinternals.com*

Оснастка Performance

Мы часто ссылаемся на этот инструмент, доступный через папку Administrative Tools (Администрирование) в меню Start (Пуск) или через Control Panel (Панель управления). Оснастка Performance (Производительность) предназначена для мониторинга системы, просмотра журналов, в которых регистрируются значения счетчиков производительности, и оповещения при достижении заданных пороговых значений тех или иных счетчиков. Говоря об оснастке Performance, мы подразумеваем лишь ее функцию системного мониторинга.

Оснастка Performance способна сообщить о том, как работает система, гораздо больше, чем любая другая, отдельно взятая утилита. Она предусматривает сотни счетчиков для различных объектов. По каждому счетчику можно получить краткое описание. Чтобы увидеть описание, выберите счетчик в окне Add Counters (Добавить счетчики) и щелкните кнопку Explain (Объяснение). Или откройте справочный файл Performance Counter Reference с компакт-диска «Ресурсы Windows». Информацию о том, как интерпретировать показания счетчиков для устранения «узких мест» в системе или для планирования пропускной способности сервера, см. раздел «Performance Monitoring» в книге «Windows 2000 Server Operations Guide» из набора Windows 2000 Server Resource Kit. Для Windows XP и Windows Server 2003 см. документацию Performance Counters Reference в Windows Server 2003 Resource Kit.

Заметьте, что все счетчики производительности Windows доступны программным путем. Краткое описание соответствующих компонентов см. в разделе «HKEY_PERFORMANCE_DATA» главы 4.

Windows Support Tools

Windows Support Tools включают около 40 утилит, полезных в администрировании систем на базе Windows и устранении неполадок в них. Многие из этих утилит раньше были частью ресурсов Windows NT 4.

Вы можете установить Support Tools, запустив Setup.exe из папки \Support\Tools в дистрибутиве любого издания Windows. Support Tools одинаковы в Windows 2000 Professional, Server, Advanced Server и Datacenter Server, а для Windows XP, равно как и для Windows Server 2003, существует своя версия Support Tools.

Ресурсы Windows

Ресурсы Windows (Windows Resource Kits) расширяют Support Tools, предлагая дополнительные утилиты для администрирования и поддержки систем. Утилиты Windows Server 2003 Resource Kit можно бесплатно скачать с *www.microsoft.com* (выполните поиск по ключевым словам «resource kit tools»). Их можно установить в Windows XP или Windows Server 2003.

Ресурсы Windows 2000 существуют в двух изданиях: Windows 2000 Professional Resource Kit и Windows 2000 Server Resource Kit* (самая последняя

* Последнее издание переведено на русский язык издательством «Русская Редакция» и выпущено в 2001 г. в виде серии «Ресурсы Microsoft Windows 2000 Server», которая включает 4 книги: «Сети TCP/IP», «Сопровождение сервера», «Распределенные системы» и «Межсетевое взаимодействие». — *Прим. перев.*

его версия — Supplement 1). Хотя последний набор представляет собой надмножество первого и может быть установлен на системах с Windows 2000 Professional, утилиты, входящие только в Windows 2000 Server Resource Kit, ни в одном из наших экспериментов не используются. В отличие от утилит Windows Server 2003 Resource Kit эти утилиты нельзя скачать бесплатно. Однако Windows 2000 Server Resource Kit поставляется с подписками на MSDN и TechNet.

Отладка ядра

Отладка ядра подразумевает изучение внутренних структур данных ядра и/или пошаговый проход по функциям в ядре. Это полезный способ исследования внутреннего устройства Windows, потому что он позволяет увидеть внутрисистемную информацию, недоступную при использовании каких-либо других способов, и получить более ясное представление о схеме выполнения кода внутри ядра.

Отладку ядра можно проводить с помощью разнообразных утилит: Windows Debugging Tools от Microsoft, LiveKD от *www.sysinternals.com* или SoftIce от Compuware NuMega. Прежде чем описывать эти средства, давайте рассмотрим файл, который понадобится при любом виде отладки ядра.

Символы для отладки ядра

Файлы символов (symbol files) содержат имена функций и переменных. Они генерируются компоновщиком (linker) и используются отладчиками для ссылки и отображения этих имен в сеансе отладки. Эта информация обычно не хранится в двоичном образе, потому что она не нужна при выполнении кода. То есть двоичные образы имеют меньший размер и работают быстрее. Но это означает, что вам нужно позаботиться о том, чтобы у отладчика был доступ к файлам символов, сопоставляемым с образами, на которые вы ссылаетесь в сеансе отладки.

Для изучения внутренних структур данных ядра Windows (например, списка процессов, блоков потока, списка загруженных драйверов, информации об использовании памяти и т. д.) вам понадобятся подходящие файлы символов как минимум для образа ядра, Ntoskrnl.exe. (Подробнее этот файл рассматривается в разделе «Обзор архитектуры» главы 2.) Файлы таблиц символов должны соответствовать версии образа. Так, если вы установили Windows Service Pack или какое-то оперативное исправление, то должны получить обновленные файлы символов хотя бы для образа ядра; иначе возникнет ошибка из-за неправильной контрольной суммы при попытке отладчика ядра загрузить их.

Хотя можно скачать и установить символы для разных версий Windows, обновленные символы для оперативных исправлений доступны не всегда. Самый простой способ получить подходящую версию символов для отладки — обратиться к Microsoft-серверу символов с запросом, в котором используется специальный синтаксис пути к символам, как в отладчике. Например,

следующий путь к символам заставляет средства отладки загружать требуемые символы с Интернет-сервера символов и сохранять локальную копию в папке c:\symbols:

```
srv*c:\symbols*http://msdl.microsoft.com/download/symbols
```

Подробные инструкции о том, как пользоваться сервером символов, см. в справочном файле Debugging Tools или на Web-странице *www.microsoft.com/whdc/ddk/debugging/symbols.mspx*.

Windows Debugging Tools

Пакет Windows Debugging Tools содержит дополнительные средства отладки, применяемые в этой книге для исследования внутреннего устройства Windows. Вы найдете их последние версии по ссылке *www.microsoft.com/whdc/ddk/debugging*. Эти средства можно использовать для отладки как процессов пользовательского режима, так и ядра (см. следующую врезку).

ПРИМЕЧАНИЕ Windows Debugging Tools регулярно обновляются и выпускаются независимо от версий операционной системы Windows, поэтому почаще проверяйте наличие новых версий отладочных средств.

Отладка в пользовательском режиме

Средства отладки можно подключать к процессу пользовательского режима, чтобы исследовать и/или изменять память процесса. Существует два варианта подключения к процессу:

- **Invasive (инвазивный)** Если не указано иное, то, когда вы подключаетесь к выполняемому процессу, Windows-функция DebugActiveProcess устанавливает соединение между отладчиком и отлаживаемым процессом. Это позволяет изучать и/или изменять память процесса, устанавливать точки прерывания (breakpoints) и выполнять другие отладочные действия. В Windows 2000 при завершении отладчика закрывается и отлаживаемый процесс. Однако в Windows XP отладчик можно отключать, не уничтожая целевой процесс.
- **Noninvasive (неинвазивный)** В этом случае отладчик просто открывает процесс через функцию OpenProcess. Он не подключается к процессу как отладчик. Это позволяет изучать и/или изменять память целевого процесса, но не дает возможности устанавливать точки прерывания. Преимущество данного варианта в том, что в Windows 2000 можно закрыть отладчик, не завершая целевой процесс.

С помощью отладочных средств также можно открывать файлы дампов процессов пользовательского режима. Что представляют собой эти файлы, поясняется в главе 3 в разделе по диспетчеризации исключений.

Microsoft предлагает отладчики ядра в двух версиях: командной строки (Kd.exe) и с графическим пользовательским интерфейсом (Windbg.exe). Оба

инструмента предоставляют одинаковый набор команд, так что выбор конкретной утилиты определяется сугубо личными пристрастиями. С помощью этих средств вы можете вести отладку ядра в трех режимах.

- Откройте файл дампа, полученный в результате краха системы с Windows (подробнее о таких дампах см. главу 14).
- Подключитесь к работающей системе и изучите ее состояние (или поставьте точки прерывания, если вы отлаживаете код драйвера устройства). Эта операция требует двух компьютеров — целевого и управляющего. Целевой считается отлаживаемая система, а управляющей — та, в которой выполняется отладчик. Целевая система может быть либо локальной (соединенной с управляющей нуль-модемным кабелем или по IEEE 1394), либо удаленной (соединенной по модему). Вы должны загрузить целевую систему со спецификатором /DEBUG, или нажать при загрузке клавишу F8 и выбрать Debug Mode, или добавить соответствующую запись в файл Boot.ini.
- В случае Windows XP и Windows Server 2003 подключитесь к локальной системе и изучите ее состояние. Это называется *локальной отладкой ядра*. Чтобы инициировать такую отладку ядра, выберите в меню File команду Kernel Debug, перейдите на вкладку Local и щелкните OK. Пример окна с выводом показан на рис. 1-7. Некоторые команды отладчика ядра в этом режиме не работают (например, просмотр стеков ядра и создание дампа памяти командой .dump невозможны). Однако вы можете пользоваться бесплатной утилитой LiveKd с сайта *www.sysinternals.com* в тех случаях, когда родные средства локальной отладки не срабатывают (см. следующий раздел).

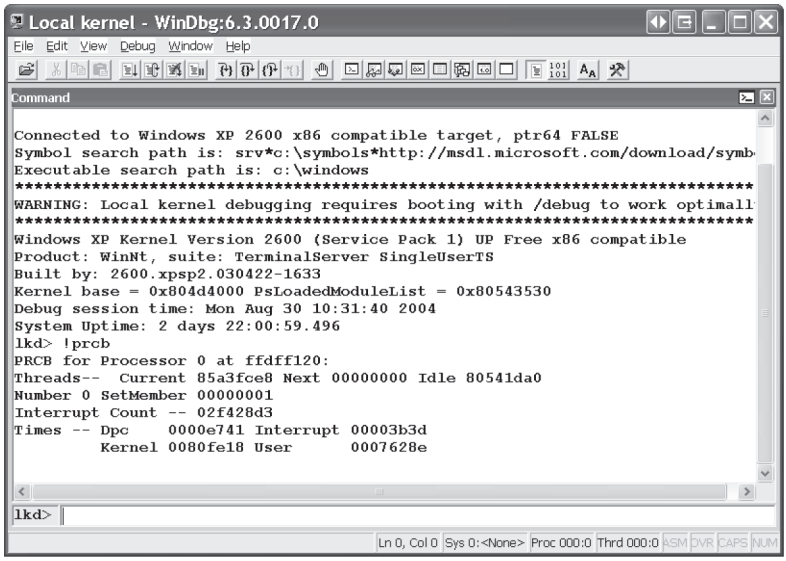


Рис. 1-7. *Локальная отладка ядра*

Подключившись в режиме отладки ядра, вы можете использовать одну из многих команд расширения отладчика (команды, которые начинаются с «!») для вывода содержимого внутренних структур данных, например потоков, процессов, пакетов запроса на ввод-вывод и информации, связанной с управлением памятью. Команды отладчика ядра и их вывод будут обсуждаться при рассмотрении соответствующей тематики. А пока добавим, что команда *dt* (display type) может форматировать свыше 400 структур ядра благодаря тому, что файлы символов ядра для Windows 2000 Service Pack 3, Windows XP и Windows Server 2003 содержат информацию о типах, которая и позволяет отладчику форматировать структуры.

ЭКСПЕРИМЕНТ: отображение информации о типах для структур ядра

Чтобы вывести список структур ядра, чья информация о типах включена в символы ядра, наберите **dt nt!_*** в отладчике ядра. Пример части вывода показан ниже:

```
lkd> dt nt!_*
          nt!_LIST_ENTRY
          nt!_LIST_ENTRY
          nt!_IMAGE_NT_HEADERS
          nt!_IMAGE_FILE_HEADER
          nt!_IMAGE_OPTIONAL_HEADER
          nt!_IMAGE_NT_HEADERS
          nt!_LARGE_INTEGER
```

Команда *dt* позволяет искать конкретные структуры по шаблонам. Например, если вы ищете имя структуры для объекта прерывания (interrupt object), введите **dt nt!_*interrupt***:

```
lkd> dt nt!_*interrupt*
          nt!_KINTERRUPT
          nt!_KINTERRUPT_MODE
```

После этого с помощью команды *dt* можно отформатировать эту структуру:

```
lkd> dt nt!_kinterrupt
nt!_KINTERRUPT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 InterruptListEntry : _LIST_ENTRY
   +0x00c ServiceRoutine   : Ptr32
   +0x010 ServiceContext   : Ptr32 Void
   +0x014 SpinLock         : Uint4B
   +0x018 TickCount        : Uint4B
   +0x01c ActualLock       : Ptr32 Uint4B
   +0x020 DispatchAddress  : Ptr32
   +0x024 Vector           : Uint4B
   +0x028 Irql             : UChar
```

см. след. стр.

```
   +0x029 SynchronizeIrql  : UChar
   +0x02a FloatingSave     : UChar
   +0x02b Connected        : UChar
   +0x02c Number           : Char
   +0x02d ShareVector      : UChar
   +0x030 Mode             : _KINTERRUPT_MODE
   +0x034 ServiceCount     : Uint4B
   +0x038 DispatchCount    : Uint4B
   +0x03c DispatchCode     : [106] Uint4B
```

Заметьте, что по умолчанию *dt* не показывает подструктуры (структуры внутри структур). Для рекурсивного прохода по подструктурам, используйте ключ **–r**. Например, указав этот ключ для отображения объекта ядра «прерывание», вы увидите формат структуры _LIST_ENTRY, хранящейся в поле InterruptListEntry:

```
lkd> dt nt!_kinterrupt -r
nt!_KINTERRUPT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 InterruptListEntry :
      +0x000 Flink            : Ptr32
         +0x000 Flink            : Ptr32 _LIST_ENTRY
         +0x004 Blink            : Ptr32 _LIST_ENTRY
      +0x004 Blink            : Ptr32
         +0x000 Flink            : Ptr32 _LIST_ENTRY
         +0x004 Blink            : Ptr32 _LIST_ENTRY
   +0x00c ServiceRoutine   : Ptr32
```

В справочном файле Windows Debugging Tools объясняется, как устанавливать и использовать отладчики ядра. Дополнительные сведения о применении отладчиков ядра, предназначенных в основном разработчикам драйверов устройств, см. в документации Windows DDK. Есть также несколько полезных статей в Knowledge Base по отладчикам ядра. Выполните поиск по ключевому слову «debugref» в Windows Knowledge Base (онлайновой базе данных технических статей) на *support.microsoft.com*.

Утилита LiveKd

LiveKd — бесплатная утилита от *www.sysinternals.com*, которая позволяет использовать стандартные отладчики ядра от Microsoft на «живой» системе — без подключения второго компьютера. Если встроенная поддержка локальной отладки ядра действует только в Windows XP и Windows Server 2003, то LiveKd обеспечивает такую отладку в Windows NT 4.0, Windows 2000, Windows XP и Windows Server 2003.

LiveKd запускается точно так же, как Windbg или Kd. Эта утилита передает любые указанные параметры командной строки выбранному вами отладчику. По умолчанию LiveKd запускает отладчик Kd. Для запуска GUI-отлад-

чика (Windbg), задайте ключ **–w**. Чтобы получить подсказку по ключам LiveKd, укажите ключ **–?**.

LiveKd предоставляет отладчику смоделированный файл аварийного дампа (crash dump), поэтому вы можете выполнять в LiveKd любые операции, поддерживаемые для аварийных дампов. Поскольку LiveKd хранит смоделированный дамп в физической памяти, отладчик ядра может попасть в такую ситуацию, в которой структуры данных находятся в рассогласованном состоянии в процессе их изменения системой. При каждом запуске отладчик получает снимок состояния системы; если вы хотите обновить этот снимок, выйдите из отладчика (командой **q**), и LiveKd спросит вас, нужно ли начать сначала. Если отладчик, выводя информацию на экран, вошел в цикл, нажмите клавиши Ctrl+C, чтобы прервать вывод, выйдите из отладчика и запустите его снова. Если он завис, нажмите клавиши Ctrl+Break, которые заставят завершить процесс отладчика. После этого вам будет предложено снова запустить отладчик.

SoftICE

Еще один инструмент, не требующий двух машин для прямой отладки ядра, — SoftICE, который можно приобрести у Compuware NuMega (*www.compuware.com*). SoftICE обладает во многом теми же возможностями, что и Windows Debugging Tools, но поддерживает переход между кодом пользовательского режима и режима ядра. Он также поддерживает DLL-модули расширения ядра Microsoft, поэтому большинство команд, описываемых нами в книге, будут работать и в SoftICE. На рис. 1-8 показан пользовательский интерфейс SoftICE, появляющийся при нажатии клавиши активизации SoftICE (по умолчанию — Ctrl+D); этот интерфейс представляет собой окно на рабочем столе системы, в которой он выполняется.

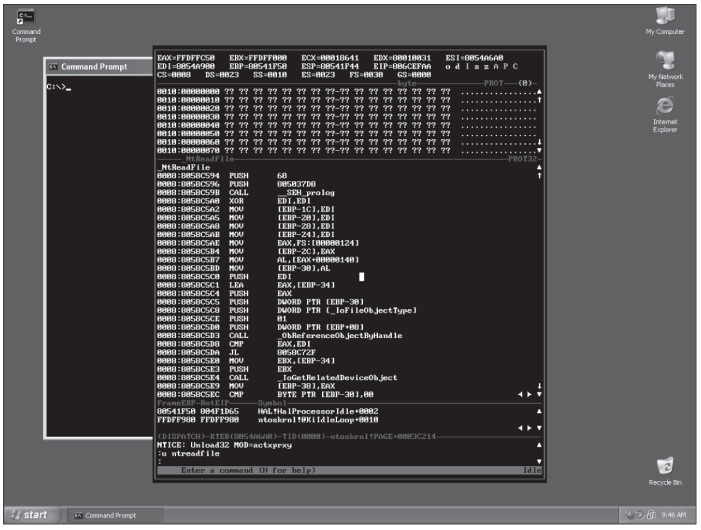


Рис. 1-8. *Интерфейс SoftICE*

Platform Software Development Kit (SDK)

Platform SDK является частью подписки на MSDN уровня Professional и выше; кроме того, его можно бесплатно скачать с *msdn.microsoft.com*. В нем содержатся документация, заголовочные файлы и библиотеки C, необходимые для компиляции и компоновки Windows-приложений. (Microsoft Visual C++ тоже поставляется с этими файлами, но их версии в Platform SDK всегда более новые и соответствуют самым последним версиям операционных систем Windows.) В Platform SDK для нас будут представлять интерес заголовочные файлы Windows API (\Program Files\Microsoft SDK\Include) и несколько утилит (Pfmon.exe, Pstat.exe, Pview.exe, Vadump.exe и Winobj.exe). Некоторые из них также поставляются с Ресурсами Windows и Support Tools. Наконец, отдельные утилиты поставляются с Platform SDK и MSDN Library как примеры исходного кода.

Device Driver Kit (DDK)

Windows DDK является частью подписки на MSDN уровня Professional и выше, но в отличие от Platform SDK его нельзя скачать бесплатно (впрочем, можно заказать CD-ROM за минимальную цену). Документация Windows DDK включается в MSDN Library.

Хотя DDK нацелен на разработчиков драйверов устройств, он представляет собой богатый источник информации о внутреннем устройстве Windows. Например, в главе 9 мы даем описание архитектуры подсистемы ввода-вывода, модели драйверов и структур данных базовых драйверов устройств, но не вдаемся в детали соответствующих функций ядра. А в документации Windows DDK исчерпывающе описаны все внутрисистемные функции и драйверы устройств.

Кроме документации в DDK входят заголовочные файлы, определяющие ключевые внутренние структуры данных, константы и интерфейсы многих внутрисистемных подпрограмм (в частности, обратите внимание на файлы Ntddk.h и Wdm.h). Эти файлы очень полезны в исследовании внутренних структур данных Windows с помощью отладчика ядра, так как мы даем лишь обобщенное описание внутренних структур, а в заголовочных файлах можно найти все подробности о каждом поле таких структур. В DDK также детально поясняются некоторые структуры данных (вроде заголовков для диспетчера объектов, блоки ожидания, события, мутанты, семафоры и др.).

Поэтому, если вы хотите поглубже покопаться в подсистеме ввода-вывода и в модели драйверов, читайте документацию DDK (особенно руководства Kernel-Mode Driver Architecture Design Guide и Reference). Еще один превосходный источник — книга Уолта Они (Walt Oney) «Programming the Microsoft Windows Driver Model, Second Edition» (Microsoft Press).

Утилиты Sysinternals

Во многих экспериментах мы используем свободно распространяемые утилиты, которые можно скачать с *www.sysinternals.com*. Большинство этих ути-

лит написано Марком Руссиновичем, соавтором этой книги. К наиболее популярным утилитам относятся Process Explorer, Filemon и Regmon. Многие из этих утилит требуют установки и запуска драйверов устройств, работающих в режиме ядра, а значит, вам понадобятся полномочия администратора.

Резюме

В этой главе вы познакомились с ключевыми техническими концепциями и терминами Windows, которые будут использоваться во всей книге. Вы также получили первое представление о многих полезных инструментах, позволяющих изучать внутренние структуры данных Windows. Теперь вы готовы вместе с нами приступить к исследованию внутреннего устройства системы. Мы начнем с общего обзора архитектуры системы и ее основных компонентов.

ГЛАВА 2

Архитектура системы

Теперь, познакомившись с необходимыми терминами, понятиями и инструментами, мы можем рассмотреть задачи, которые ставились при разработке операционной системы Microsoft Windows. В этой главе описывается общая архитектура системы: ключевые компоненты, принципы их взаимодействия и контекст выполнения. Чтобы получить базовое представление о внутреннем устройстве Windows, давайте сначала обсудим требования и цели, обусловившие структуру и спецификацию этой системы.

Требования и цели проекта

Характеристики Windows NT в 1989 году определялись следующими требованиями. Операционная система должна:

- быть истинно 32-разрядной, реентерабельной, поддерживать вытесняющую многозадачность и работу с виртуальной памятью;
- работать на разных аппаратных платформах;
- хорошо масштабироваться в системах с симметричной мультипроцессорной обработкой;
- быть распределенной вычислительной платформой, способной выступать в роли как клиента сети, так и сервера;
- поддерживать большинство существующих 16-разрядных приложений MS-DOS и Microsoft Windows 3.1;
- отвечать требованиям правительства к соответствию POSIX 1003.1;
- отвечать требованиям правительства и промышленности к безопасности операционных систем;
- обеспечивать простоту адаптации к глобальному рынку за счет поддержки Unicode.

Для создания системы, соответствующей предъявленным требованиям, нужно было принять тысячи решений. Поэтому перед командой разработчиков Windows NT на начальном этапе проекта были поставлены следующие цели.

- **Расширяемость** Код должен быть написан так, чтобы системы можно было легко наращивать и модифицировать по мере изменения потребностей рынка.

- **Переносимость** Система должна работать на разных аппаратных архитектурах и обладать способностью к сравнительно легкому переносу на новые аппаратные архитектуры, если на рынке возникнет такая потребность.
- **Отказоустойчивость и надежность** Система должна быть защищенной как от внутренних сбоев, так и от внешних деструктивных действий. У приложений не должно быть возможности нарушить работу операционной системы или других приложений.
- **Совместимость** Хотя Windows NT должна расширить существующую технологию, ее пользовательский интерфейс и API должны быть совместимы с предыдущими версиями Windows и MS-DOS. Она также должна уметь взаимодействовать с другими системами вроде UNIX, OS/2 и NetWare.
- **Производительность** С учетом ограничений, налагаемых поставленными целями, система должна быть максимально быстрой и отзывчивой независимо от аппаратной платформы.

По мере изучения деталей внутренней структуры Windows вы увидите, насколько успешно были реализованы все эти требования и цели. Но сначала мы рассмотрим общую модель Windows и сравним ее с другими современными операционными системами.

Модель операционной системы

В большинстве многопользовательских операционных систем приложения отделены от собственно операционной системы: код ее ядра выполняется в привилегированном режиме процессора (называемом *режимом ядра*), который обеспечивает доступ к системным данным и оборудованию. Код приложений выполняется в непривилегированном режиме процессора (называемом *пользовательским*) с неполным набором интерфейсов, ограниченным доступом к системным данным и без прямого доступа к оборудованию. Когда программа пользовательского режима вызывает системный сервис, процессор перехватывает вызов и переключает вызывающий поток в режим ядра. По окончании работы системного сервиса операционная система переключает контекст потока обратно в пользовательский режим и продолжает его выполнение.

Windows, как и большинство UNIX-систем, является монолитной операционной системой — в том смысле, что бóльшая часть ее кода и драйверов использует одно и то же пространство защищенной памяти режима ядра. Это значит, что любой компонент операционной системы или драйвер устройства потенциально способен повредить данные, используемые другими компонентами операционной системы.

Основана ли Windows на микроядре?

Хотя некоторые объявляют ее таковой, Windows не является операционной системой на основе микроядра в классическом понимании этого термина. В подобных системах основные компоненты операционной системы (диспетчеры памяти, процессов, ввода-вывода) выполняются как отдельные процессы в собственных адресных пространствах и представляют собой надстройки над примитивными сервисами микроядра. Пример современной системы с архитектурой на основе микроядра — операционная система Mach, разработанная в Carnegie Mellon University. Она реализует крошечное ядро, которое включает сервисы планирования потоков, передачи сообщений, виртуальной памяти и драйверов устройств. Все остальное, в том числе разнообразные API, файловые системы и поддержка сетей, работает в пользовательском режиме. Однако в коммерческих реализациях на основе микроядра Mach код файловой системы, поддержки сетей и управления памятью выполняется в режиме ядра. Причина проста: системы, построенные строго по принципу микроядра, непрактичны с коммерческой точки зрения из-за слишком низкой эффективности.

Означает ли тот факт, что большая часть Windows работает в режиме ядра, ее меньшую надежность в сравнении с операционными системами на основе микроядра? Вовсе нет. Рассмотрим следующий сценарий. Допустим, в коде файловой системы имеется ошибка, которая время от времени приводит к краху системы. Ошибка в коде режима ядра (например, в диспетчере памяти или файловой системы) скорее всего вызовет полный крах традиционной операционной системы. В истинной операционной системе на основе микроядра подобные компоненты выполняются в пользовательском режиме, поэтому теоретически ошибка приведет лишь к завершению процесса соответствующего компонента. Но на практике такая ошибка все равно вызовет крах системы, так как восстановление после сбоя столь критически важного процесса невозможно.

Все эти компоненты операционной системы, конечно, полностью защищены от сбойных приложений, поскольку такие программы не имеют прямого доступа к коду и данным привилегированной части операционной системы (хотя и способны вызывать сервисы ядра). Эта защита — одна из причин, по которым Windows заслужила репутацию отказоустойчивой и стабильной операционной системы в качестве сервера приложений и платформы рабочих станций, обеспечивающей быстродействие основных системных сервисов вроде поддержки виртуальной памяти, файлового ввода-вывода, работы с сетями и доступа к общим файлам и принтерам.

Компоненты Windows режима ядра также построены на принципах объектно-ориентированного программирования (ООП). Так, для получения информации о каком-либо компоненте они, как правило, не обращаются к его

структурам данных. Вместо этого для передачи параметров, доступа к структурам данных и их изменения используются формальные интерфейсы.

Однако, несмотря на широкое использование объектов, представляющих разделяемые системные ресурсы, Windows не является объектно-ориентированной системой в строгом понимании этого термина. Бóльшая часть системного кода написана на C в целях переносимости и из-за широкой распространенности средств разработки на C. В этом языке нет прямой поддержки конструкций и механизмов ООП вроде динамического связывания типов данных, полиморфных функций или наследования классов.

Обзор архитектуры

Теперь обратимся к ключевым компонентам системы, составляющим ее архитектуру. Упрощенная версия этой архитектуры показана на рис. 2-1. Учтите, что упрощенная схема не отражает всех деталей архитектуры (например, здесь не показаны уровни сетевых компонентов и различных типов драйверов устройств).

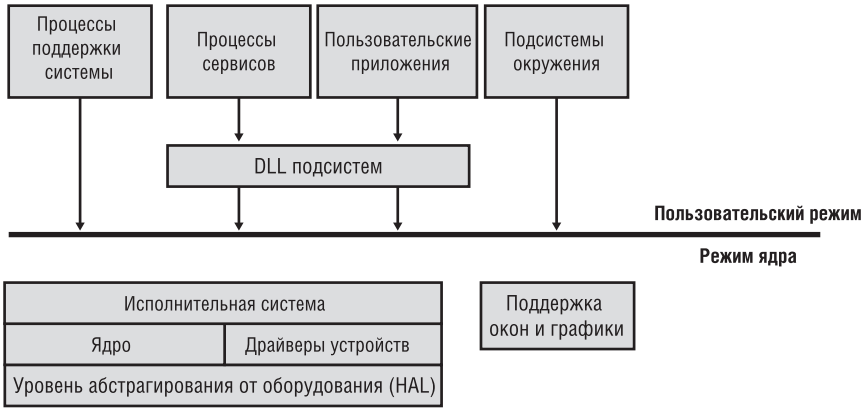


Рис. 2-1. *Упрощенная схема архитектуры Windows*

На рис. 2-1 прежде всего обратите внимание на линию, разделяющую те части Windows, которые выполняются в режиме ядра и в пользовательском режиме. Прямоугольники над этой линией соответствуют процессам пользовательского режима, а компоненты под ней — сервисам режима ядра. Как говорилось в главе 1, потоки пользовательского режима выполняются в защищенных адресных пространствах процессов (хотя при выполнении в режиме ядра они получают доступ к системному пространству). Таким образом, процессы поддержки системы, сервисов, приложений и подсистем окружения имеют свое адресное пространство.

Существует четыре типа пользовательских процессов:

- фиксированные *процессы поддержки системы* (system support processes) — например, процесс обработки входа в систему и диспетчер сеансов, не являющиеся сервисами Windows (т. е. не запускаемые диспетчером управления сервисами);

- *процессы сервисов* (service processes) — носители Windows-сервисов вроде Task Scheduler и Spooler. Многие серверные приложения Windows, например Microsoft SQL Server и Microsoft Exchange Server, тоже включают компоненты, выполняемые как сервисы;
- *пользовательские приложения* (user applications) — бывают шести типов: для 32-разрядной Windows, 64-разрядной Windows, 16-разрядной Windows 3.1, 16-разрядной MS-DOS, 32-разрядной POSIX и 32-разрядной OS/2;
- *подсистемы окружения* (environment subsystems) — реализованы как часть поддержки среды операционной системы, предоставляемой пользователям и программистам. Изначально Windows NT поставлялась с тремя подсистемами окружения: Windows, POSIX и OS/2. Последняя была изъята в Windows 2000. Что касается Windows XP, то в ней исходно поставляется только подсистема Windows — улучшенная подсистема POSIX доступна как часть бесплатного продукта Services for UNIX.

Обратите внимание на прямоугольник «DLL подсистем», расположенный на рис. 2-1 под прямоугольниками «процессы сервисов» и «пользовательские приложения». В Windows пользовательские приложения не могут вызывать родные сервисы операционной системы напрямую, вместо этого они работают с одной или несколькими *DLL подсистем*. Их назначение заключается в трансляции документированных функций в соответствующие внутренние (и обычно недокументированные) вызовы системных сервисов Windows. Трансляция может осуществляться как с помощью сообщения, посылаемого процессу подсистемы окружения, обслуживающему пользовательское приложение, так и без него.

Windows включает следующие компоненты режима ядра.

- *Исполнительная система* (executive) Windows, содержащая базовые сервисы операционной системы, которые обеспечивают управление памятью, процессами и потоками, защиту, ввод-вывод и взаимодействие между процессами.
- *Ядро* (kernel) Windows, содержащее низкоуровневые функции операционной системы, которые поддерживают, например, планирование потоков, диспетчеризацию прерываний и исключений, а также синхронизацию при использовании нескольких процессоров. Оно также предоставляет набор процедур и базовых объектов, применяемых исполнительной системой для реализации структур более высокого уровня.
- *Драйверы устройств* (device drivers), в состав которых входят драйверы аппаратных устройств, транслирующие пользовательские вызовы функций ввода-вывода в запросы, специфичные для конкретного устройства, а также сетевые драйверы и драйверы файловых систем.
- *Уровень абстрагирования от оборудования* (hardware abstraction layer, HAL), изолирующий ядро, драйверы и исполнительную систему Windows от специфики оборудования на данной аппаратной платформе (например, от различий между материнскими платами).

- *Подсистема поддержки окон и графики* (windowing and graphics system), реализующая функции графического пользовательского интерфейса (GUI), более известные как Windows-функции модулей USER и GDI. Эти функции обеспечивают поддержку окон, элементов управления пользовательского интерфейса и отрисовку графики.

В таблице 2-1 перечислены имена файлов основных компонентов Windows. (Вы должны знать их, потому что в дальнейшем мы будем ссылаться на некоторые системные файлы по именам.) Каждый из этих компонентов подробно рассматривается в этой и последующих главах.

Таблица 2-1. *Основные системные файлы Windows*

Имя файла	Компоненты
Ntoskrnl.exe	Исполнительная система и ядро
Ntkrnlpa.exe (только 32-разрядные системы)	Исполнительная система и ядро с поддержкой механизма Physical Address Extension (PAE), позволяющего адресовать 64 Гб физической памяти
Hal.dll	Уровень абстрагирования от оборудования
Win32sk.sys	Часть подсистемы Windows, работающая в режиме ядра
Ntdll.dll	Внутренние функции поддержки и интерфейсы (stubs) диспетчера системных сервисов с функциями исполнительной системы
Kernel32.dll, Advapi32.dll, User32.dll, Gdi32.dll	Основные DLL подсистемы Windows

Прежде чем детально рассматривать эти компоненты, давайте проясним, как достигается переносимость Windows между различными аппаратными платформами.

Переносимость

Windows рассчитана на разные аппаратные платформы, включая как CISC-системы Intel, так и RISC-системы. Windows NT первого выпуска поддерживала архитектуры x86 и MIPS. Спустя некоторое время была добавлена поддержка Alpha AXP производства DEC (DEC была приобретена Compaq, а позднее произошло слияние компаний Compaq и Hewlett Packard). (Хотя Alpha AXP был 64-разрядным процессором, Windows NT работала с ним в 32-разрядном режиме. В ходе разработки Windows 2000 была создана ее 64-разрядная версия специально под Alpha AXP, но в свет она так и не вышла.) В Windows NT 3.51 ввели поддержку четвертой процессорной архитектуры — Motorola PowerPC. В связи с изменениями на рынке необходимость в поддержке MIPS и PowerPC практически отпала еще до начала разработки Windows 2000. Позднее Compaq отозвала поддержку архитектуры Alpha AXP, и в Windows 2000 осталась поддержка лишь архитектуры x86. В самые последние выпуски — Windows XP и Windows Server 2003 — добавлена поддержка трех семейств 64-разрядных процессоров: Intel Itanium IA-64, AMD x86-64 и Intel 64-bit Extension Technology (EM64T) для x86 (эта архитектура совме-

стима с архитектурой AMD x86-64, хотя есть небольшие различия в поддерживаемых командах). Последние два семейства процессоров называются *системами с 64-разрядными расширениями* и в этой книге обозначаются как x64. (Как 32-разрядные приложения выполняются в 64-разрядной Windows, объясняется в главе 3.)

Переносимость Windows между системами с различной аппаратной архитектурой и платформами достигается главным образом двумя способами.

- Windows имеет многоуровневую структуру. Специфичные для архитектуры процессора или платформы низкоуровневые части системы вынесены в отдельные модули. Благодаря этому высокоуровневая часть системы не зависит от специфики архитектур и аппаратных платформ. Ключевые компоненты, обеспечивающие переносимость операционной системы, — ядро (содержится в файле Ntoskrnl.exe) и уровень абстрагирования от оборудования (HAL) (содержится в файле Hal.dll). Функции, специфичные для конкретной архитектуры (переключение контекста потоков, диспетчеризация ловушек и др.), реализованы в ядре. Функции, которые могут отличаться на компьютерах с одинаковой архитектурой (например, в системах с разными материнскими платами), реализованы в HAL. Еще один компонент, содержащий большую долю кода, специфичного для конкретной архитектуры, — диспетчер памяти (memory manager), но если рассматривать систему в целом, такого кода все равно немного.
- Подавляющее большинство компонентов Windows написано на C и лишь часть из них — на C++. Язык ассемблера применяли только при создании частей системы, напрямую взаимодействующих с системным оборудованием (например, при написании обработчика ловушек прерываний) или требующих исключительного быстродействия (скажем, при переключении контекста). Ассемблерный код имеется не только в ядре и HAL, но и в составе некоторых других частей операционной системы: процедур, реализующих взаимоблокировку, механизма вызова локальных процедур (LPC), части подсистемы Windows, выполняемой в режиме ядра, и даже в некоторых библиотеках пользовательского режима (например, в коде запуска процессов в Ntdll.dll — системной библиотеке, о которой будет рассказано в этой главе несколько позже).

Симметричная многопроцессорная обработка

Многозадачность (multitasking) — механизм операционной системы, позволяющий использовать один процессор для выполнения нескольких потоков. Однако истинно одновременное выполнение, например, двух потоков возможно, только если на компьютере установлено два процессора. При многозадачности система лишь создает видимость одновременного выполнения множества потоков, тогда как многопроцессорная система действительно выполняет сразу несколько потоков — по одному на каждом процессоре.

Как уже говорилось в начале этой главы, одной из ключевых целей разработки Windows была поддержка многопроцессорных компьютерных сис-

тем. Windows является операционной системой, поддерживающей *симметричную многопроцессорную обработку* (symmetric multiprocessing, SMP). В этой модели нет главного процессора; операционная система, как и пользовательские потоки, может выполняться на любом процессоре. Кроме того, все процессоры используют одну и ту же память. При *асимметричной многопроцессорной обработке* (asymmetric multiprocessing, ASMP) система, напротив, выбирает один из процессоров для выполнения кода ядра операционной системы, а другие процессоры выполняют только пользовательский код. Различия между этими двумя моделями показаны на рис. 2-2.

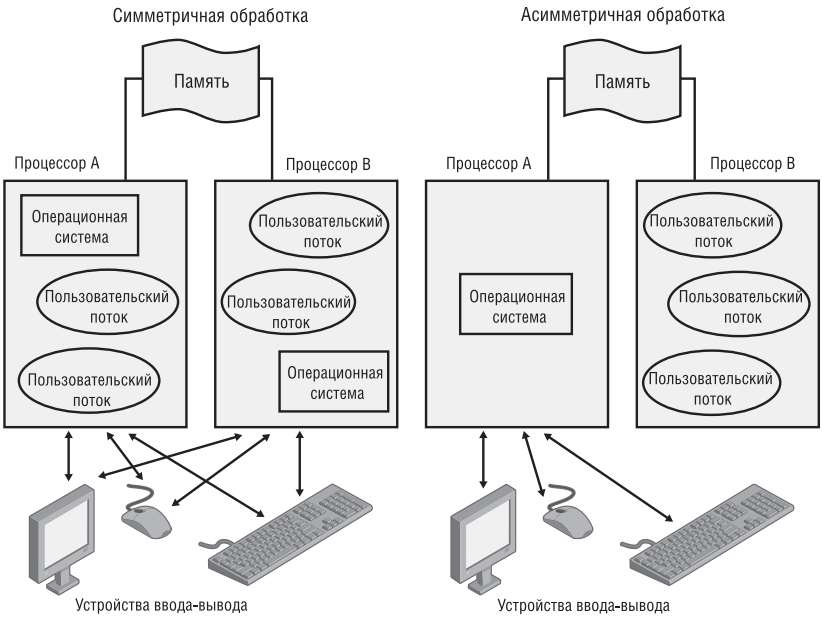


Рис. 2-2. *Симметричная и асимметричная многопроцессорная обработка*

Windows XP и Windows Server 2003 поддерживают два новых типа многопроцессорных систем: логические процессоры (hyperthreading) и NUMA (Non-Uniform Memory Architecture). Об этом кратко рассказывается в абзаце ниже. (Полное описание поддержки планирования потоков для таких систем см. в разделе по планированию потоков в главе 6.)

Логические процессоры — это технология, созданная Intel; благодаря ей на одном физическом процессоре может быть несколько логических. Каждый логический процессор имеет свое состояние, но исполняющее ядро (execution engine) и набортный кэш (onboard cache) являются общими. Это позволяет одному из логических процессоров продолжать работу, пока другой логический процессор занят (например, обработкой прерывания, которая не дает потокам выполняться на этом логическом процессоре). Алгоритмы планирования в Windows XP были оптимизированы под компьютеры с такими процессорами.

В NUMA-системах процессоры группируются в блоки, называемые узлами (nodes). В каждом узле имеются свои процессоры и память, и он соединяется с остальными узлами специальной шиной. Windows в NUMA-системе по-прежнему работает как SMP-система, в которой все процессоры имеют доступ ко всей памяти, — просто доступ к памяти, локальной для узла, осуществляется быстрее, чем к памяти в других узлах. Система стремится повысить производительность, выделяя потокам время на процессорах, которые находятся в том же узле, что и используемая память. Она также пытается выделять память в пределах узла, но при необходимости выделяет память и из других узлов.

Хотя Windows изначально разрабатывалась для поддержки до 32 процессоров, многопроцессорной модели не свойственны никакие внутренние особенности, которые ограничивали бы число используемых процессоров до 32. Просто это число легко представить битовой маской с помощью машинного 32-разрядного типа данных. И действительно, 64-разрядные версии Windows поддерживают до 64 процессоров, потому что размер слова на 64-разрядных процессорах равен 64 битам.

Реальное число поддерживаемых процессоров зависит от конкретного выпуска Windows (см. таблицы 2-3 и 2-4). Это число хранится в параметре реестра HKLM\SYSTEM\CurrentControlSet\Control\Session\Manager\LicensedProcessors. (Учтите, что модификация этого параметра считается нарушением условий лицензионного соглашения на программное обеспечение, да и для увеличения числа поддерживаемых процессоров требуется нечто большее, чем простое изменение данного параметра.)

Для большей производительности ядро и HAL имеют одно- и многопроцессорную версии. В случае Windows 2000 это относится к шести ключевым системным файлам (см. примечание ниже), а в 32-разрядных Windows XP и Windows Server 2003 — только к трем (см. таблицу 2-2). В 64-разрядных системах Windows ядра PAE нет, поэтому одно- и многопроцессорные системы отличаются лишь ядром и HAL.

Соответствующие файлы выбираются и копируются в локальный каталог \Windows\System32 на этапе установки. Чтобы определить, какие файлы были скопированы, см. файл \Windows\Repair\Setup.log, где перечисляются все файлы, копировавшиеся на локальный системный диск, и каталоги на дистрибутивном носителе, откуда они были взяты.

Таблица 2-2. *Системные файлы, специфичные для одно- и многопроцессорных систем*

Имя файла на системном диске	Имя однопроцессорной версии на дистрибутивном носителе	Имя многопроцессорной версии на дистрибутивном носителе
Ntoskrnl.exe	Ntoskrnl.exe	Ntkrnlmp.exe
Ntkrnlpa.exe (ядро PAE; только 32-разрядные системы)	Ntkrnlpa.exe в \Windows\<arch>\Driver.cab	Ntkrpamp.exe в \Windows\<arch>\Driver.cab
Hal.dll	Зависит от типа системы (см. список HAL в таблице 2-6)	Зависит от типа системы (см. список HAL в таблице 2-6)
Только Windows 2000		
Win32k.sys	\I386\UNIPROC\Win32k.sys	Win32k.sys в \I386\Driver.cab
Ntdll.dll	\I386\UNIPROC\Ntdll.dll	\I386\Ntdll.dll
Kernel32.dll	\I386\UNIPROC\Kernel32.dll	\I386\Kernel32.dll

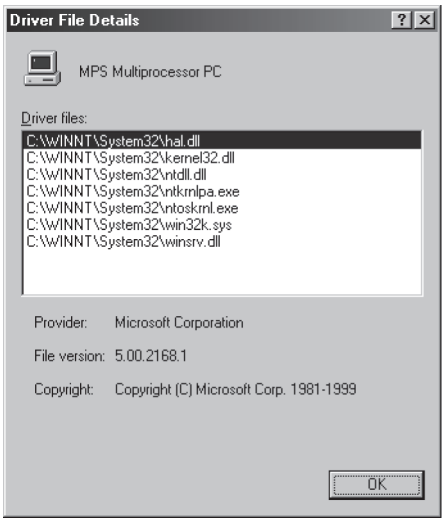
ПРИМЕЧАНИЕ В папке \I386\UNIPROC в дистрибутиве Windows 2000 находится файл Winsrv.dll. Хотя он помещен в папку UNIPROC, название которой указывает на однопроцессорную версию, на самом деле для одно- и многопроцессорных систем существует только одна версия этого образа.

ЭКСПЕРИМЕНТ: поиск файлов поддержки многопроцессорных систем в Windows 2000

Вы можете убедиться в том, что для многопроцессорной 32-разрядной системы Windows 2000 используются другие файлы, просмотрев сведения о драйверах для Computer (Компьютер) в Device Manager (Диспетчер устройств).

1. Откройте окно свойств системы, дважды щелкнув System (Система) в окне Control Panel (Панель управления) или щелкнув правой кнопкой мыши My Computer (Мой компьютер) на рабочем столе и выбрав из контекстного меню команду Properties (Свойства).
2. Перейдите на вкладку Hardware (Оборудование).
3. Щелкните кнопку Device Manager (Диспетчер устройств).
4. Раскройте объект Computer (Компьютер).
5. Дважды щелкните дочерний узел объекта Computer.
6. Откройте вкладку Driver (Драйвер).
7. Щелкните кнопку Driver Details (Сведения о драйверах).

В многопроцессорной системе вы должны увидеть диалоговое окно, показанное ниже.



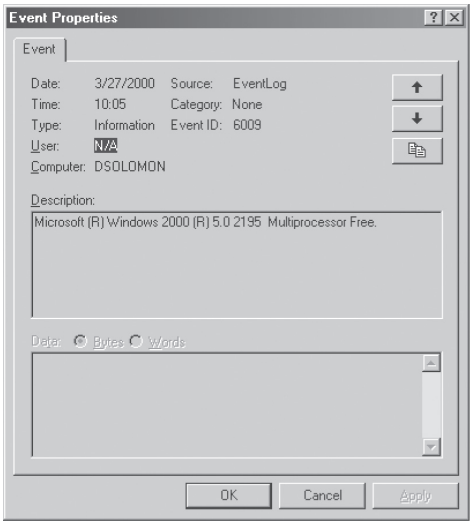
Специальные версии этих ключевых системных файлов для однопроцессорных систем созданы для максимального повышения производительности. Синхронизация работы нескольких процессоров — задача принципиально более сложная, и благодаря «однопроцессорным» версиям системных файлов устраняются издержки этой синхронизации, которая в однопроцессорных системах (а они составляют подавляющее большинство систем под управлением Windows) не нужна.

Интересно, что «однопроцессорная» и «многопроцессорная» версии Ntoskrnl создаются за счет условной компиляции одного и того же исходного кода, а «однопроцессорные» версии Ntdll.dll и Kernel32.dll для Windows 2000 требуют замены машинных x86-команд LOCK и UNLOCK, используемых для синхронизации множества потоков, командой NOP (которая ничего не делает).

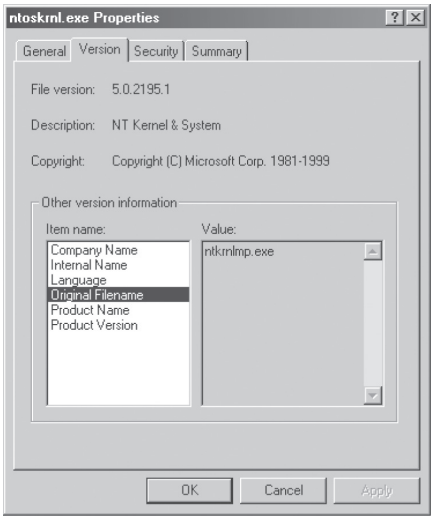
Остальные системные файлы Windows (включая все утилиты, библиотеки и драйверы устройств) одинаковы как в многопроцессорных, так и в однопроцессорных системах. При разработке нового программного обеспечения — Windows-приложения или драйвера устройства — вы должны учитывать этот подход и тестировать свое программное обеспечение как в одно-, так и в многопроцессорных системах.

ЭКСПЕРИМЕНТ: определение текущей версии Ntoskrnl

В Windows 2000 и выше нет утилиты, показывающей, с какой версией Ntoskrnl вы работаете. Однако при каждой загрузке в журнале системы регистрируется, какая версия ядра запускается — одно- или многопроцессорная, отладочная или конечная (см. следующую иллюстрацию). Выберите из меню Start (Пуск) команду Programs (Программы), затем Administrative Tools (Администрирование) и Event Viewer (Просмотр событий). Далее выберите System Log (Журнал системы) и дважды щелкните запись с кодом события 6009 — она создается при загрузке системы.



Эта запись не содержит сведений о том, загружена ли PAE-версия образа ядра, поддерживающая более 4 Гб физической памяти (Ntkrnlpa.exe). Однако вы можете узнать это, проверив значение параметра SystemStartOptions в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control. Кроме того, при загрузке PAE-версии ядра параметру PhysicalAddressExtension в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management присваивается значение, равное 1.



Есть и другой способ определить, установлена ли многопроцессорная версия Ntoskrnl (или Ntkrnlpa): запустите Windows Explorer (Проводник), в каталоге \Windows\System32 щелкните правой кнопкой мыши файл Ntoskrnl.exe и выберите из контекстного меню команду Pro-

см. след. стр.

perties (Свойства). Перейдите на вкладку Version (Версия) и выберите свойство Original Filename (Исходное имя файла). Если вы работаете с многопроцессорной версией, то увидите диалоговое окно, показанное на предыдущей странице.

Наконец, просмотрев файл \Windows\Repair\Setup.log, можно точно выяснить, какие файлы ядра и HAL были выбраны при установке.

Масштабируемость

Масштабируемость (scalability) — одна из ключевых целей многопроцессорных систем. Для корректного выполнения в SMP-системах операционная система должна строго соответствовать определенным требованиям. Решить проблемы конкуренции за ресурсы и другие вопросы в многопроцессорных системах сложнее, чем в однопроцессорных, и это нужно учитывать при разработке системы. Некоторые особенности Windows оказались решающими для ее успеха как многопроцессорной операционной системы:

- способность выполнять код операционной системы на любом доступном процессоре и на нескольких процессорах одновременно;
- несколько потоков одного процесса можно параллельно выполнять на разных процессорах;
- тонкая синхронизация внутри ядра (спин-блокировки, спин-блокировки с очередями и др.; см. главу 3), драйверов устройств и серверных процессов позволяет выполнять больше компонентов на нескольких процессорах одновременно;
- механизмы вроде портов завершения ввода-вывода (см. главу 9), облегчающие эффективную реализацию многопоточных серверных процессов, хорошо масштабируемых в многопроцессорных системах.

Масштабируемость ядра Windows со временем улучшалась. Например, в Windows Server 2003 имеются очереди планирования, индивидуальные для каждого процессора, что дает возможность планировать потоки параллельно на нескольких машинах. О планировании потоков в многопроцессорных системах см. главу 6, а о синхронизации в таких системах — главу 3.

Различия между клиентскими и серверными версиями

Windows поставляется в клиентских и серверных версиях. В Windows 2000 клиентская версия называется Windows 2000 Professional. Существует также три серверных версии Windows 2000: Windows 2000 Server, Advanced Server и Datacenter Server.

У Windows XP шесть клиентских версий: Windows XP Home Edition, Windows XP Professional, Windows XP Starter Edition, Windows XP Tablet PC Edition, Windows XP Media Center Edition и Windows XP Embedded. Последние три являются надмножествами Windows XP Professional и в книге детально не рассматриваются, так как все они построены на том же базовом коде, что и Windows XP Professional.

Windows Server 2003 выпускается в шести разновидностях: Windows Server 2003 Web Edition, Standard Edition, Small Business Server, Storage Server, Enterprise Edition и Datacenter Edition.

Эти версии различаются по следующим параметрам:

- числу поддерживаемых процессоров;
- объему поддерживаемой физической памяти;
- возможному количеству одновременных сетевых соединений (например, в клиентской версии допускается максимум 10 одновременных соединений со службой доступа к общим файлам и принтерам);
- наличием в выпусках Server сервисов, не входящих в Professional (например, служб каталогов, поддержкой кластеризации и многопользовательской службы терминала).

Эти различия для Windows 2000 суммируются в таблице 2-3. Та же информация, но применительно к Windows XP и Windows Server 2003 дана в таблице 2-4. Детальное сравнение различных выпусков Windows Server 2003 см. по ссылке *http://www.microsoft.com/windowsserver2003/evaluation/features/compareeditions.mspx*.

Таблица 2-3. *Различия между Windows 2000 Professional и Server*

Выпуск	Число процессоров	Объем физической памяти, Гб
Windows 2000 Professional	2	4
Windows 2000 Server	4	4
Windows 2000 Advanced Server	8	8
Windows 2000 Datacenter Server	32	64

Таблица 2-4. *Различия между Windows XP и Windows Server 2003*

	Число процессоров (32-разрядная версия)	Объем физической памяти, Гб (32-разрядная	Число процессоров (64-разрядная версия)	Объем физической памяти, Гб (версии для для Itanium)	Объем физической памяти, Гб (версии для x64)
Windows XP Home Edition	1	4	Неприменимо	Неприменимо	Неприменимо
Windows XP Professional	2	4	2	16	16
Windows Server 2003 Web Edition	2	2	Неприменимо	Неприменимо	Неприменимо
Windows Server 2003 Small Business Server	2	2	Неприменимо	Неприменимо	Неприменимо
Windows Server 2003 Standard Edition	4	4	Неприменимо	Неприменимо	Неприменимо

см. след. стр.

Таблица 2-4. (*окончание*)

	Число процессоров (32-разрядная версия)	Объем физической памяти, Гб (32-разрядная	Число процессоров (64-разрядная версия)	Объем физической памяти, Гб (версии для для Itanium)	Объем физической памяти, Гб (версии для x64)
Windows Server 2003 Enterprise Edition	8	32	8	64	64
Windows Server 2003 Datacenter Edition	32	128 (на x64); 64 (на x86)	64	512 (1024 Гб в SP1)	Неприменимо

Хотя существует несколько клиентских и серверных выпусков операционной системы Windows, у них общий набор базовых системных файлов, в том числе: ядро, Ntoskrnl.exe (а также версия PAE, Ntkrnlpa.exe), библиотеки HAL, драйверы, основные системные утилиты и DLL. Эти файлы идентичны для всех выпусков Windows 2000.

ПРИМЕЧАНИЕ Windows XP была первым клиентским выпуском кодовой базы Windows NT, который поставляется без соответствующих серверных версий. Вместо этого разработки продолжались, и примерно год спустя после выхода Windows XP была выпущена Windows Server 2003. Таким образом, базовые системные файлы Windows XP и Windows Server 2003 не идентичны. Однако они не столь значимы (и во многих случаях компоненты не изменялись).

Итак, если образ ядра для Windows 2000 Professional и Windows 2000 Server одинаков (и сходен для Windows XP и Windows Server 2003), то как же система определяет, какой именно выпуск загружается? Для этого она проверяет значения параметров ProductType и ProductSuite в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\ProductOptions. Параметр ProductType используется, чтобы отличить клиентскую систему от серверной (любого выпуска). Список допустимых значений этого параметра приведен в таблице 2-5. Результат проверки помещается в глобальную системную переменную *MmProductType*, значение которой может быть запрошено драйвером устройства через функцию *MmIsThisAnNtAsSystem* режима ядра, описанную в документации Windows DDK.

Таблица 2-5. *Значения параметра реестра ProductType*

Выпуск Windows	Значение ProductType
Windows 2000 Professional, Windows XP Professional, Windows XP Home Edition	WinNT
Windows Server (контроллер домена)	LanmanNT
Windows Server (только сервер)	ServerNT

Другой параметр, ProductSuite, позволяет различать серверные версии Windows (Standard, Enterprise, Datacenter Server и др.), а также Windows XP Home от Windows XP Professional. Для проверки текущего выпуска Windows пользовательские программы вызывают Windows-функцию *VerifyVersionInfo*, описанную в Platform SDK. Драйверы могут вызвать функцию *RtlGetVersion* режима ядра, документированную в Windows DDK.

Итак, если базовые файлы в целом одинаковы для клиентских и серверных версий, то чем же отличается их функционирование? Серверные системы оптимизированы для работы в качестве высокопроизводительных серверов приложений, а клиентские, несмотря на поддержку серверных возможностей, — для персональных систем. Так, некоторые решения по выделению ресурсов (например, о числе и размере системных пулов памяти, количестве внутрисистемных рабочих потоков и размере системного кэша данных) при загрузке принимаются по-разному, в зависимости от типа продукта. Политика принятия таких решений, как обслуживание диспетчером памяти запросов системы и процессов на выделение памяти, у серверной и клиентской версий тоже различается. В равной мере это относится и к особенностям планирования потоков по умолчанию (детали см. в главе 6). Существенные отличия в функционировании этих двух продуктов будут отмечены в соответствующих главах. Любые материалы в нашей книге, если явно не указано иное, относятся к обеим версиям — клиентской и серверной.

Проверочный выпуск

Специальная отладочная версия Windows 2000 Professional, Windows XP Professional или Windows Server 2003 называется *проверочным выпуском* (checked build). Она доступна только подписчикам MSDN уровня Professional (или выше). Проверочный выпуск представляет собой перекомпилированный исходный код Windows, для которого флаг «DBG» (заставляющий включать код отладки и трассировки этапа компиляции) был установлен как TRUE. Кроме того, чтобы облегчить восприятие машинного кода, отключается обработка двоичных файлов, при которой структура кода оптимизируется для большего быстродействия (см. раздел «Performance-Optimized Code» в справочном файле Debugging Tools).

Проверочный выпуск предназначен главным образом разработчикам драйверов устройств, поскольку эта версия выполняет более строгую проверку ошибок при вызове функций режима ядра драйверами устройств или другим системным кодом. Например, если драйвер (или какой-то иной код режима ядра) неверно вызывает системную функцию, контролирующую передаваемые параметры, то при обнаружении этой проблемы система останавливается, предотвращая повреждение структур данных и возможный крах.

ЭКСПЕРИМЕНТ: определяем, является ли данная система проверочным выпуском

Встроенной утилиты, которая позволяла бы увидеть, с каким выпуском вы имеете дело — проверочным или готовым, нет. Однако эта информация доступна через свойство «Debug» WMI-класса (Windows Management Instrumentation) Win32_OperatingSystem. Следующий сценарий на Visual Basic отображает содержимое этого свойства:

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:" _
 & "{impersonationLevel=impersonate}!\\" & _
 strComputer & "\root\cimv2")
Set colOperatingSystems = objWMIService.ExecQuery _
 ("SELECT * FROM Win32_OperatingSystem"S)
For Each objOperatingSystem in colOperatingSystems
 Wscript.Echo "Caption: " & objOperatingSystem.Caption
 Wscript.Echo "Debug: " & objOperatingSystem.Debug
 Wscript.Echo "Version: " & objOperatingSystem.Version
Next
```

Чтобы опробовать его в действии, наберите код этого сценария и сохраните как файл. Вот пример вывода:

```
C:\>cscript osversion.vbs
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001.
All rights reserved.

Caption: Microsoft Windows XP Professional
Debug: False
Version: 5.1.2600
```

Эта система не является проверочным выпуском, так как флаг Debug равен False.

Значительная часть дополнительного кода в собранных таким образом двоичных файлах является результатом работы макроса ASSERT, определенного в заголовочном файле Ntddk.h, который входит в состав DDK. Этот макрос проверяет некое условие (например, правильность структуры данных или параметра) и, если значение выражения получается равным FALSE, вызывает функцию *RtlAssert* режима ядра, которая в свою очередь обращается к *DbgPrint*, передающей текст отладочного сообщения в буфер отладочных сообщений (debug message buffer). Если отладчик ядра подключен, это сообщение выводится на экран, а за ним автоматически появляется запрос к пользователю, какое действие следует предпринять (игнорировать, завершить процесс или поток и т. д.). Если система загружена без отладчика ядра (в отсутствие ключа /DEBUG в файле Boot.ini) и этот отладчик сейчас не подключен, неудачный тест ASSERT вызовет крах системы. Список тестов

ASSERT, выполняемых некоторыми вспомогательными процедурами ядра, см. в разделе «Checked Build ASSERTs» документации Windows DDK.

ПРИМЕЧАНИЕ Сравнив файл Ntoskrnl.exe с Ntkrnlmp.exe или Ntkrnlpa.exe с Ntkrpamp.exe в проверочной версии системы, вы убедитесь, что они идентичны и являются «многопроцессорными» версиями соответствующих файлов. Иначе говоря, в проверочной версии системы нет отладочных вариантов файлов для однопроцессорных систем.

Проверочный выпуск также полезен системным администраторам, так как в нем можно включить детальную трассировку для определенных компонентов. (Подробные инструкции см. в статье 314743 «HOWTO: Enable Verbose Debug Tracing in Various Drivers and Subsystems» в Microsoft Knowledge Base.) Вывод такой трассировки посылается в буфер отладочных сообщений с помощью функции DbgPrint, о которой мы уже упоминали. Для просмотра отладочных сообщений к целевой системе можно подключить отладчик ядра (что потребует загрузки целевой системы в отладочном режиме), использовать команду *!dbgprint* в процессе локальной отладки ядра или применить утилиту Dbgview.exe с сайта *www.sysinternals.com*.

Для использования возможностей отладочной версии операционной системы необязательно устанавливать весь проверочный выпуск. Можно просто скопировать проверочную версию образа ядра (Ntoskrnl.exe) и соответствующий HAL (Hal.dll) в обычную систему. Преимущество этого подхода в том, что он позволяет тщательно протестировать драйверы устройств и другой код ядра, не устанавливая медленнее работающие версии всех компонентов системы. О том, как это сделать, см. раздел «Installing Just the Checked Operating System and HAL» в документации Windows DDK. Поскольку Microsoft не поставляет проверочный выпуск Windows 2000 Server, вы можете применить этот способ и получить проверочную версию ядра в системе Windows 2000 Server.

Наконец, проверочная версия пригодится и для тестирования кода пользовательского режима, но только в том смысле, что в проверочной версии системы устанавливаются другие интервалы ожидания (тайминги). (Это связано с тем, что в ядре выполняются дополнительные проверки, а сами компоненты компилируются без оптимизации.) В таких условиях часто проявляются ошибки, связанные с синхронизацией нескольких потоков приложения.

Ключевые компоненты системы

Теперь, ознакомившись с высокоуровневой архитектурой Windows, копнем поглубже и рассмотрим роль каждого ключевого компонента системы. На рис. 2-3 отражена более подробная схема системной архитектуры Windows. Заметьте, что на ней все равно не показаны некоторые компоненты (в частности, компоненты сетевой поддержки, о которых пойдет речь в главе 13).

Основные элементы этой схемы детально описываются в последующих главах. В главе 3 рассказывается об основных механизмах управления,

используемых системой (в том числе о диспетчере объектов, прерываниях и т. п.), в главе 5 — о процессах запуска и завершения Windows, а в главе 4 — о таких механизмах управления, как реестр, процессы сервисов и Windows Management Instrumentation (WMI). В остальных главах не менее подробно поясняется внутреннее устройство и функционирование ключевых элементов — процессов, потоков, подсистемы управления памятью, защиты, диспетчера ввода-вывода, диспетчера кэша, файловой системы NTFS, сетевой поддержки и др.

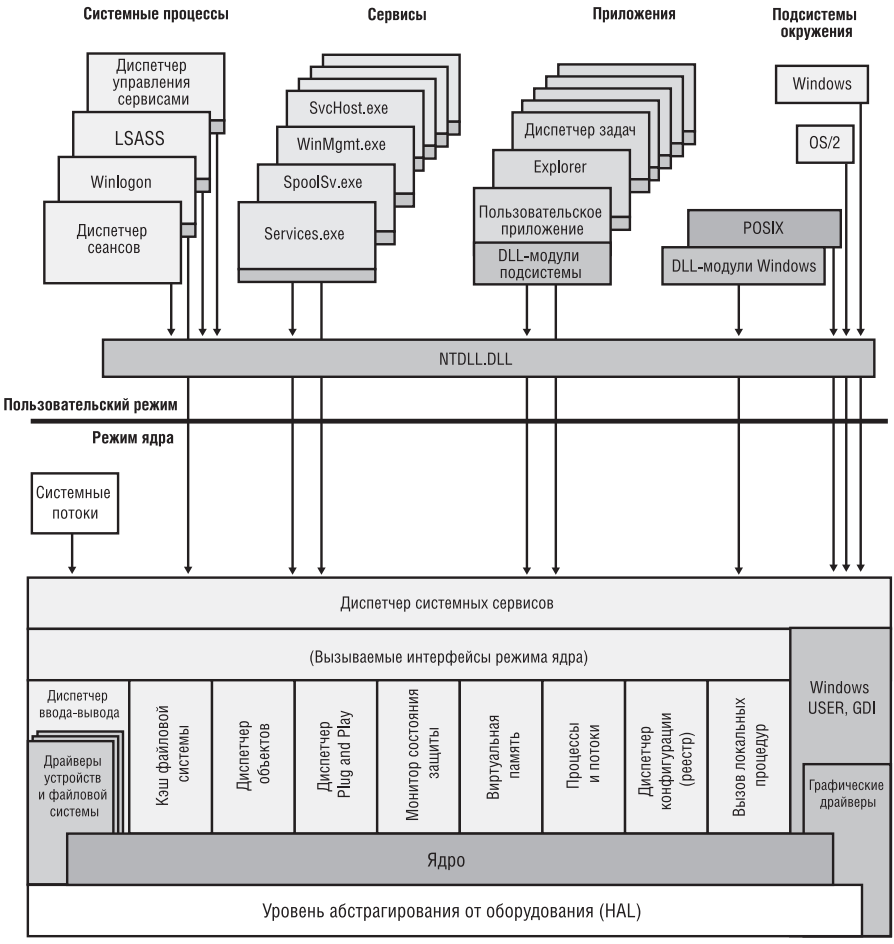


Рис. 2-3. *Архитектура Windows*

Подсистемы окружения и их DLL

Как показано на рис. 2-3, в Windows имеется три подсистемы окружения: OS/2, POSIX и Windows. Как мы уже говорили, подсистема OS/2 была удалена в Windows 2000. Начиная с Windows XP, базовая подсистема POSIX не поставляется с Windows, но ее гораздо более совершенную версию можно получить бесплатно как часть продукта Services for UNIX.

Подсистема Windows отличается от остальных двух тем, что без нее Windows работать не может (эта подсистема обрабатывает все, что связано с клавиатурой, мышью и экраном, и нужна даже на серверах в отсутствие интерактивных пользователей). Фактически остальные две подсистемы запускаются только по требованию, тогда как подсистема Windows работает всегда.

Стартовая информация подсистемы хранится в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\SubSystems. Значения параметров в этом разделе показаны на рис. 2-4.

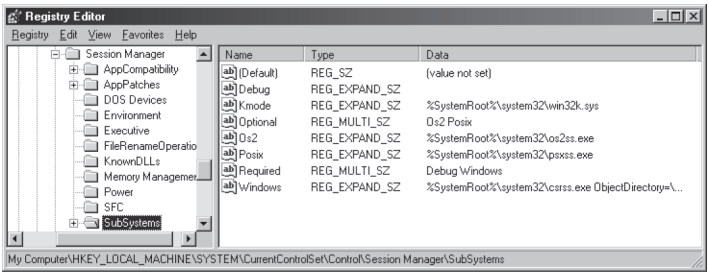


Рис. 2-4. *Просмотр стартовой информации Windows в Registry Editor*

Значением параметра Required является список подсистем, загружаемых при запуске системы. Параметр состоит из двух строк: Windows и Debug. В параметре Windows указывается спецификация файла подсистемы Windows, Csrss.exe (аббревиатура от Client/Server Run-Time Subsystem; см. примечание ниже). Параметр Debug остается незаполненным (он используется для внутреннего тестирования) и не выполняет никаких функций. Параметр Optional указывает, что подсистемы OS/2 и POSIX запускаются по требованию. Параметр Kmode содержит имя файла той части подсистемы Windows, которая работает в режиме ядра, — Win32k.sys (об этом файле чуть позже).

Подсистемы окружения предоставляют прикладным программам некое подмножество базовых сервисов исполнительной системы Windows. Каждая подсистема обеспечивает доступ к разным подмножествам встроенных сервисов Windows. Это значит, что приложения, созданные для одной подсистемы, могут выполнять операции, невозможные в другой подсистеме. Так, Windows-приложения не могут использовать POSIX-функцию *fork*.

Каждый исполняемый образ (EXE) принадлежит одной — и только одной — подсистеме. При запуске образа код, отвечающий за создание процесса, получает тип подсистемы, указанный в заголовке образа, и уведомляет соответствующую подсистему о новом процессе. Тип указывается специфика-

тором /SUBSYSTEM в команде *link* в Microsoft Visual C++; его можно просмотреть с помощью утилиты Exetype, входящей в состав ресурсов Windows.

ПРИМЕЧАНИЕ Процесс подсистемы Windows назван Csrss.exe потому, что в Windows NT все подсистемы изначально предполагалось выполнять как потоки внутри единственного общесистемного процесса. Когда подсистемы POSIX и OS/2 были выделены в собственные процессы, имя файла процесса подсистемы Windows осталось прежним.

Смешивать вызовы функций разных подсистем нельзя. Иными словами, приложения POSIX могут вызывать только сервисы, экспортируемые подсистемой POSIX, а приложения Windows — лишь сервисы, экспортируемые подсистемой Windows. Как вы еще убедитесь, это ограничение послужило одной из причин, по которой исходная подсистема POSIX, реализующая весьма ограниченный набор функций (только POSIX 1003.1), не стала полезной средой для переноса в нее UNIX-приложений.

Мы уже говорили, что пользовательские приложения не могут вызывать системные сервисы Windows напрямую. Вместо этого они обращаются к DLL подсистем. Эти DLL предоставляют документированный интерфейс между программами и вызываемой ими подсистемой. Так, DLL подсистемы Windows (Kernel32.dll, Advapi32.dll, User32.dll и Gdi32.dll) реализуют функции Windows API. DLL подсистемы POSIX (Psxdll.dll) реализует POSIX API.

ЭКСПЕРИМЕНТ: определение типа подсистемы, для которой предназначен исполняемый файл

Вы можете определить, для какой подсистемы предназначен исполняемый файл с помощью утилиты Exetype из набора ресурсов Windows или утилиты Dependency Walker (Depends.exe), входящей в состав Windows Support Tools и Platform SDK. Попробуем, например, выяснить тип подсистемы для двух принципиально разных Windows-образов: Notepad.exe (простого текстового редактора) и Cmd.exe (поддержки командной строки Windows).

```
C:\>exetype \windows\system32\notepad.exe
File "\windows\system32\notepad.exe" is of the following type:
    Windows NT
    32 bit machine
    Built for the Intel 80386 processor
    Runs under the Windows GUI subsystem

C:\>exetype \windows\system32\cmd.exe
File "\windows\system32\cmd.exe" is of the following type:
    Windows NT
    32 bit machine
    Built for the Intel 80386 processor
    Runs under the Windows character-based subsystem
```

Это показывает, что Notepad является GUI-программой, а Cmd — консольной, или программой текстового режима. Хотя вывод утилиты Exetype сообщает о наличии двух разных подсистем для GUI- и консольных программ, на самом деле существует лишь одна подсистема Windows. Кроме того, Windows не поддерживает процессор Intel 386 (или 486, если это имеет какое-то значение) — текст сообщений, выводимых программой Exetype, просто не обновили.

При вызове приложением одной из функций DLL подсистемы возможно одно из трех.

- Функция полностью реализована в пользовательском режиме внутри DLL подсистемы. Иначе говоря, никаких сообщений процессу подсистемы окружения не посылается, и вызова сервисов исполнительной системы Windows не происходит. После выполнения функции в пользовательском режиме результат возвращается вызвавшей ее программе. Примерами таких функций могут служить *GetCurrentProcess* (всегда возвращает –1, значение, определенное для ссылки на текущий процесс во всех функциях, связанных с процессами) и *GetCurrentProcessId* (идентификатор процесса не меняется в течение его срока жизни, поэтому его можно получить из кэша, что позволяет избежать переключения в режим ядра).
- Функция требует одного или более вызовов исполнительной системы Windows. Например, Windows-функции *ReadFile* и *WriteFile* обращаются к внутренним недокументированным сервисам ввода-вывода — соответственно к *NtReadFile* и *NtWriteFile*.
- Функция требует выполнения каких-либо операций в процессе подсистемы окружения (такие процессы, работающие в пользовательском режиме, отвечают за обслуживание клиентских приложений, выполняемых под их контролем). В этом случае подсистеме окружения выдается клиент-серверный запрос через сообщение с требованием выполнить какую-либо операцию, и DLL подсистемы, прежде чем вернуть управление вызвавшей программе, ждет соответствующего ответа.

Некоторые функции вроде *CreateProcess* и *CreateThread* могут требовать выполнения как второго, так и третьего пункта.

Хотя структура Windows позволяет поддерживать несколько независимых подсистем окружения, с практической точки зрения было бы неудобно включать в состав каждой подсистемы свой код для обработки окон и отображения ввода-вывода. Это привело бы к дублированию системных функций и в конечном счете негативно отразилось бы на объеме и производительности системы. Поскольку главной подсистемой была Windows, разработчики решили разместить эти базовые функции именно в ней. Так что другие подсистемы для отображения ввода-вывода вызывают соответствующие сервисы Windows. (Кстати, посмотрев на тип подсистемы в заголовках их файлов, вы убедитесь, что фактически они являются исполняемыми файлами Windows.)

Теперь поближе познакомимся с каждой подсистемой окружения.

Подсистема Windows

Эта подсистема состоит из следующих основных элементов.

- Процесса подсистемы окружения (Csrss.exe), предоставляющего:
  - поддержку консольных (текстовых) окон;
  - поддержку создания и удаления процессов и потоков;
  - частичную поддержку процессов 16-разрядной виртуальной DOS-машины (VDM);
  - множество других функций, например *GetTempFile*, *DefineDosDevice*, *ExitWindowsEx*, а также несколько функций поддержки естественных языков.
- Драйвера режима ядра (Win32k.sys), включающего:
  - диспетчер окон, который управляет отрисовкой и выводом окон на экран, принимает ввод с клавиатуры, мыши и других устройств, а также передает пользовательские сообщения приложениям;
  - Graphics Device Interface (GDI), который представляет собой библиотеку функций для устройств графического вывода. В GDI входят функции для манипуляций с графикой и отрисовки линий, текста и фигур.
- DLL-модулей подсистем (Kernel32.dll, Advapi32.dll, User32.dll и Gdi32.dll), транслирующих вызовы документированных функций Windows API в вызовы соответствующих (и в большинстве своем недокументированных) сервисов режима ядра из Ntoskrnl.exe и Win32k.sys.
- Драйверов графических устройств, представляющих собой специфичные для конкретного оборудования драйверы графического дисплея, принтера и минипорт-драйверы видеоплат.

Для формирования элементов управления пользовательского интерфейса на экране, например окон и кнопок, приложения могут вызывать стандартные функции USER. Диспетчер окон передает эти вызовы GDI, а тот — драйверам графических устройств, где они форматируются для дисплея. Драйвер дисплея работает в паре с соответствующим минипорт-драйвером видеоплаты, обеспечивая полную поддержку видео.

GDI предоставляет набор стандартных функций двухмерной графики, которые позволяют приложениям, не имеющим представления о графических устройствах, обращаться к ним. GDI-функции играют роль посредника между приложениями и драйверами дисплея и принтера. GDI интерпретирует запросы приложений на вывод графики и посылает соответствующие запросы драйверам. Он также предоставляет приложениям стандартный унифицированный интерфейс для использования самых разнообразных устройств графического вывода. Этот интерфейс обеспечивает независимость кода приложений от конкретного оборудования и его драйверов. GDI выдает свои запросы с учетом возможностей конкретного устройства, часто разделяя запрос на несколько частей для обработки. Так, некоторые устройства сами умеют формировать эллипсы, а другие требуют от GDI интер-

претировать эллипсы как набор пикселов с определенными координатами. Подробнее об архитектуре подсистемы вывода графики и драйвере дисплея см. раздел «Design Guide» в книге «Graphics Drivers» из Windows DDK.

До Windows NT 4 диспетчер окон и графические сервисы были частью процесса подсистемы Windows пользовательского режима. В Windows NT 4 основная часть кода, ответственного за обработку окон и графики, перенесена из контекста процесса подсистемы Windows в набор вызываемых сервисов, выполняемых в режиме ядра (в файл Win32k.sys). Этот перенос был осуществлен в основном для повышения общей производительности системы. Отдельный серверный процесс, содержащий графическую подсистему, требовал многочисленных переключений контекста потоков и процессов, что отнимало большое количество тактов процессора и значительные ресурсы памяти, даже несмотря на высокую оптимизацию исходной архитектуры этой подсистемы.

Например, каждый клиентский поток обслуживается парным серверным потоком в процессе подсистемы Windows, ожидающем запросов от клиентского потока. Для передачи сообщений между потоками используется специальный механизм взаимодействия между процессами, так называемый быстрый LPC (fast LPC). В отличие от обычного переключения контекста потоков передача данных между парными потоками через быстрый LPC не вызывает в ядре события перепланирования, что позволяет серверному потоку выполняться в течение оставшегося кванта времени клиентского потока (вне очереди, определенной планировщиком). Более того, для быстрой передачи больших структур данных, например битовых карт, используются разделяемые буферы памяти, и клиенты получают прямой доступ (только для чтения) к ключевым структурам данных сервера, а это сводит к минимуму необходимость в частом переключении контекста между клиентами и сервером Windows.

GDI-операции выполняются в пакетном режиме. При этом серия графических объектов, запрошенных Windows-приложениями, не обрабатывается сервером и не прорисовывается на устройстве вывода до тех пор, пока не будет заполнена вся очередь GDI. Размер очереди можно установить через Windows-функцию *GdiSetBatchLimit*. В любой момент все объекты из очереди можно сбросить вызовом функции *GdiFlush*. С другой стороны, неизменяемые свойства и структуры данных GDI после получения от процессов подсистемы Windows кэшируются клиентскими процессами для ускорения последующего доступа к ним.

Однако, несмотря на такую оптимизацию, общая производительность системы по-прежнему не соответствовала требованиям приложений, интенсивно работающих с графикой. Очевидным решением проблемы стал перевод подсистемы поддержки окон и графики в режим ядра, что позволило избежать потребности в дополнительных потоках и связанных с ними переключениями контекста. Кроме того, как только приложения вызывают диспетчер окон и GDI, эти подсистемы теперь получают прямой доступ к

компонентам исполнительной системы Windows без перехода из пользовательского режима в режим ядра и обратно. Прямой доступ особенно важен в случае вызова GDI через видеодрайверы, когда взаимодействие с видеооборудованием требует высокой пропускной способности.

Так что же остается в той части процесса подсистемы Windows, которая работает в пользовательском режиме? Поскольку консольные программы не перерисовывают окна, все операции по отрисовке и обновлению консольных и текстовых окон проводятся именно этой частью Windows. Увидеть ее деятельность несложно: просто откройте окно командной строки и перетащите поверх него другое окно. Вы увидите, что процесс подсистемы Windows начинает расходовать процессорное время, перерисовывая консольное окно. Кроме поддержки консольных окон, только небольшая часть Windows-функций посылает сообщения процессу подсистемы Windows. К ним относятся функции, отвечающие за создание и завершение процессов и потоков, назначение букв сетевым дискам, создание временных файлов. Как правило, Windows-приложение нечасто переключает (если вообще переключает) контекст в процесс подсистемы Windows.

Не пострадала ли стабильность Windows от перевода USER и GDI в режим ядра?

Некоторые интересуются, не повлияет ли на стабильность системы перевод такой значительной части кода в режим ядра. Но риск снижения стабильности системы минимален. Дело в том, что до Windows NT 4 (равно как и в настоящее время) ошибка вроде нарушения доступа (access violation) в процессе подсистемы Windows пользовательского режима (Csrss.exe) приводила к краху системы, потому что процесс подсистемы Windows был и остается жизненно важным для функционирования всей системы. Поскольку структуры данных, определяющие окна на экране, содержатся именно в этом процессе, его гибель приводит к уничтожению пользовательского интерфейса. Однако даже при функционировании Windows в качестве сервера без интерактивных процессов система не могла бы работать без Csrss, поскольку серверные процессы иногда используют оконные сообщения для контроля внутреннего состояния приложений. Так что в Windows ошибки вроде нарушения доступа в том же коде, только выполняемом в режиме ядра, просто быстрее приводят к краху — исключения в режиме ядра требуют прекращения работы системы.

Правда, теоретически появляется другая опасность. Поскольку этот код выполняется в режиме ядра, ошибка (например, применение неверного указателя) может повредить защищенные структуры данных режима ядра. До Windows NT 4 это могло привести к нарушению доступа, так как запись в страницы режима ядра из пользовательского режима не разрешается. Но результатом стал бы крах системы. Теперь же при выполнении кода в режиме ядра запись на какую-либо страни-

цу памяти по неверному указателю не обязательно вызовет немедленный крах системы. Но, если при этом будут повреждены какие-то структуры данных, крах скорее всего произойдет. Тем не менее возникает риск, что из-за такого указателя будет повреждена не структура данных, а буфер памяти, и это приведет к возврату пользовательской программе или записи на диск неверных данных.

Существует еще одно негативное последствие перевода графических драйверов в режим ядра. Ранее некоторые части графического драйвера выполнялись в Csrss, а остальные части — в режиме ядра. Теперь весь драйвер работает только в режиме ядра. Так как не все драйверы поддерживаемых Windows графических устройств разрабатываются Microsoft, она тесно сотрудничает с производителями оборудования, чтобы гарантировать разработку ими надежных и эффективных драйверов. Все поставляемые с системой драйверы тестируются так же тщательно, как и другие компоненты исполнительной системы.

Наконец, важно понимать, что такая схема (при которой подсистема поддержки окон и графики выполняется в режиме ядра) не является принципиально рискованной. Идентичный подход используется для многих других драйверов устройств (например, сетевых карт и жестких дисков). Все эти драйверы, выполняемые в режиме ядра, никогда не снижали надежности Windows NT.

Некоторые распространяют измышления насчет снижения эффективности вытесняющей многозадачности Windows из-за перевода диспетчера окон и GDI в режим ядра. Теория, которая стоит за этой точкой зрения, — увеличивается время, затрачиваемое на дополнительную обработку Windows в режиме ядра. Это мнение возникло в результате ошибочного понимания архитектуры Windows. Действительно, во многих других операционных системах, формально поддерживающих вытесняющую многозадачность, планировщик никогда не вытесняет потоки, выполняемые в режиме ядра, или вытесняет, но лишь в отдельных ситуациях. Однако в Windows любые потоки, выполняемые в режиме ядра, планируются и вытесняются так же, как и потоки пользовательского режима, — код исполнительной системы полностью реентерабелен. Помимо многих других соображений, это просто необходимо для достижения высокого уровня масштабируемости системы на оборудовании с поддержкой SMP.

Другое направление спекуляций касалось снижения масштабируемости SMP в результате уже описанных изменений. Теоретические обоснования были такими: раньше во взаимодействии между приложением и диспетчером окон или GDI участвовали два потока: один — в приложении и один — в Csrss.exe. Поэтому в SMP-системах, где эти потоки могут выполняться параллельно, пропускная способность возрастает. Это свидетельствует о непонимании технологий, применявшихся до Windows NT 4. В большинстве случаев клиентские приложения вызывают процесс подсистемы Windows синхронно, т. е. клиент-

см. след. стр.

ский поток полностью блокируется в ожидании обработки вызова серверным потоком и возобновляется только после этого. Так что никакой параллелизм в SMP-системах недостижим. Это явление легко наблюдать в SMP-системах на примере приложений, интенсивно работающих с графикой. При этом обнаружится, что в двухпроцессорной системе каждый процессор загружен на 50%; также легко заметить единственный поток Csrss, отделенный от потока приложения. Действительно, поскольку два потока тесно взаимодействуют и находятся в сходном состоянии, для поддержания синхронизации процессорам приходится постоянно сбрасывать кэш. Именно по этой причине однопоточные графические приложения в SMP-системах под управлением Windows NT 3.51 обычно выполняются медленнее, чем в однопроцессорных системах.

В результате изменений, внесенных в Windows NT 4, удалось повысить пропускную способность SMP-систем для приложений, интенсивно использующих диспетчер окон и GDI, — особенно когда в приложении работает более одного потока. При наличии двух потоков приложения на двухпроцессорной машине под управлением Windows NT 3.51 за процессорное время конкурируют в общей сложности четыре потока (два — в приложении и два — в Csrss). Хотя в каждый момент к выполнению готовы, как правило, лишь два потока, их рассогласованность ведет к потере локальности ссылок и синхронизации кэша. Это происходит скорее всего из-за переключения потоков приложения с одного процессора на другой. В Windows NT 4 каждый из двух потоков приложения по сути имеет собственный процессор, а механизм автоматической привязки потоков в Windows пытается постоянно выполнять данный поток на одном и том же процессоре, максимально увеличивая локальность ссылок и сводя к минимуму потребность в синхронизации кэш-памяти индивидуальных процессоров.

В заключение отметим, что повышение производительности в результате перевода диспетчера окон и GDI из пользовательского режима в режим ядра достигнуто без сколько-нибудь значимого снижения стабильности и надежности системы — даже в случае нескольких сеансов, созданных в конфигурации с поддержкой Terminal Services.

Подсистема POSIX

POSIX, название которой представляет собой аббревиатуру от «portable operating system interface based on UNIX» (переносимый интерфейс операционной системы на основе UNIX), — это совокупность международных стандартов на интерфейсы операционных систем типа UNIX. Стандарты POSIX стимулировали производителей поддерживать совместимость реализуемых ими UNIX-подобных интерфейсов, тем самым позволяя программистам легко переносить свои приложения между системами.

В Windows реализован лишь один из многих стандартов POSIX, а именно POSIX.1, который официально называется ISO/IEC 9945-1:1990, или IEEE POSIX стандарта 1003.1-1990. Этот стандарт изначально был включен в основном для соответствия требованиям правительства США, установленным во второй половине 80-х годов. В федеральном стандарте Federal Information Processing Standard (FIPS) 151-2, разработанном государственным институтом стандартов и технологий (NIST), содержится требование совместимости с POSIX 1. Windows NT 3.5, 3.51 и 4 прошли тестирование на соответствие FIPS 151-2.

Поскольку совместимость с POSIX.1 была одной из обязательных целей, в Windows включена необходимая базовая поддержка подсистемы POSIX.1 — например, функция *fork*, реализованная в исполнительной системе Windows, и поддержка файловой системой Windows жестких файловых связей (hard file links). Однако POSIX.1 определяет лишь ограниченный набор сервисов (управление процессами, взаимодействие между процессами, простой символьный ввод-вывод и т. д.), и поэтому подсистема POSIX в Windows не является полноценной средой программирования. Так как вызов функций из разных подсистем Windows невозможен, набор функций, доступный приложениям POSIX по умолчанию, строго ограничен сервисами, определяемыми POSIX.1. Смысл этих ограничений в следующем: приложение POSIX не может создать поток или окно в Windows, а также использовать RPC или сокеты.

Для преодоления этого ограничения предназначен продукт Microsoft Windows Services for UNIX, включающий (в версии 3.5) улучшенную подсистему окружения POSIX, которая предоставляет около 2000 функций UNIX и 300 инструментов и утилит в стиле UNIX. (Детали см. на *www.microsoft.com/windows/sfu/default.asp*).

Эта улучшенная подсистема POSIX реально помогает переносить UNIX-приложения в Windows. Однако, поскольку эти программы все равно связаны с исполняемыми файлами POSIX, Windows-функции им недоступны. Чтобы UNIX-приложения, переносимые в Windows, могли использовать Windows-функции, нужно приобрести специальные пакеты для переноса UNIX-программ в Windows, подобные продуктам MKS Toolkit, разработанные компанией Mortice Kern Systems Inc. (*www.mkssoftware.com*). Тогда UNIX-приложения можно перекомпилировать и заново собрать как исполняемые файлы Windows и начать постепенный переход на «родные» Windows-функции.

ЭКСПЕРИМЕНТ: наблюдаем старт подсистемы POSIX

Подсистема POSIX по умолчанию сконфигурирована на запуск в момент начала выполнения приложения POSIX. Таким образом, старт подсистемы POSIX можно наблюдать, запустив какую-нибудь программу POSIX, например одну из утилит POSIX из Windows Services for UNIX (небольшой набор утилит вы найдете и в каталоге \Apps\POSIX на компакт-диске ресурсов Windows 2000; они не устанавливаются как часть

см. след. стр.

ресурсов). Для запуска подсистемы POSIX следуйте инструкциям, приведенным ниже.

1. Откройте окно командной строки.
2. Запустите Process Explorer и убедитесь, что подсистема POSIX еще не запущена (т. е. процесса Psxss.exe в системе нет). Также убедитесь, что Process Explorer отображает список процессов как дерево (нажмите Ctrl+T).
3. Запустите POSIX-программу (например C Shell или Korn Shell, поставляемую с Windows Services for UNIX) или утилиту POSIX из ресурсов Windows 2000, например \Apps\POSIX\Ls.exe.
4. Вернитесь в Process Explorer и обратите внимание на новый процесс Psxss.exe, являющийся дочерним процессом Smss.exe (который в зависимости от выбранного интервала подсветки может какое-то время оставаться выделенным как новый процесс).

Для компиляции и сборки приложения POSIX в Windows нужны заголовочные файлы и библиотеки POSIX из Platform SDK. Исполняемые файлы POSIX связываются с библиотекой подсистемы POSIX, Psxdll.dll. Поскольку Windows по умолчанию сконфигурирована на запуск подсистемы POSIX только по требованию, при первом запуске приложения POSIX должен запуститься процесс подсистемы POSIX (Psxss.exe). Его выполнение продолжается до перезагрузки системы. (Если вы завершите процесс подсистемы POSIX, запуск приложений POSIX станет невозможен до следующей перезагрузки системы.) Приложение POSIX не выполняется самостоятельно; для него запускается специальный файл поддержки Posix.exe, создающий дочерний процесс, из которого и запускаются приложения POSIX.

Подсистема OS/2

Подсистема окружения OS/2, как и подсистема POSIX, обладает довольно ограниченной функциональностью и поддерживает лишь 16-разрядные приложения OS/2 версии 1.2 с символьным или графическим вводом-выводом. Кроме того, Windows запрещает прикладным программам прямой доступ к оборудованию и поэтому не поддерживает приложения OS/2, использующие расширенный ввод-вывод видео или включающие сегменты привилегированного ввода-вывода, которые пытаются выполнять инструкции IN/OUT (для доступа к некоторым аппаратным устройствам). Приложения, выдающие машинные команды CLI/STI, могут работать в Windows, но на время выполнения команды STI все другие приложения OS/2 в системе и потоки процессов OS/2, выдающих команды CLI, приостанавливаются.

Как показано на рис. 2-5, подсистема OS/2, использующая 32-разрядное виртуальное адресное пространство Windows, может предоставить приложениям OS/2 версии 1.2 до 512 Мб памяти, снимая тем самым исходное ограничение этой версии на объем адресуемой памяти (до 16 Мб).

Мозаичная область (tiled area) — это 512 Мб заранее резервируемого виртуального адресного пространства, откуда передается и куда возвращается память, выделяемая под сегменты, которыми пользуются 16-разрядные приложения. Для каждого процесса подсистема OS/2 ведет таблицу локальных дескрипторов (local descriptor table, LDT), в которой сегменты разделяемой памяти занимают один и тот же LDT-слот для всех процессов OS/2.

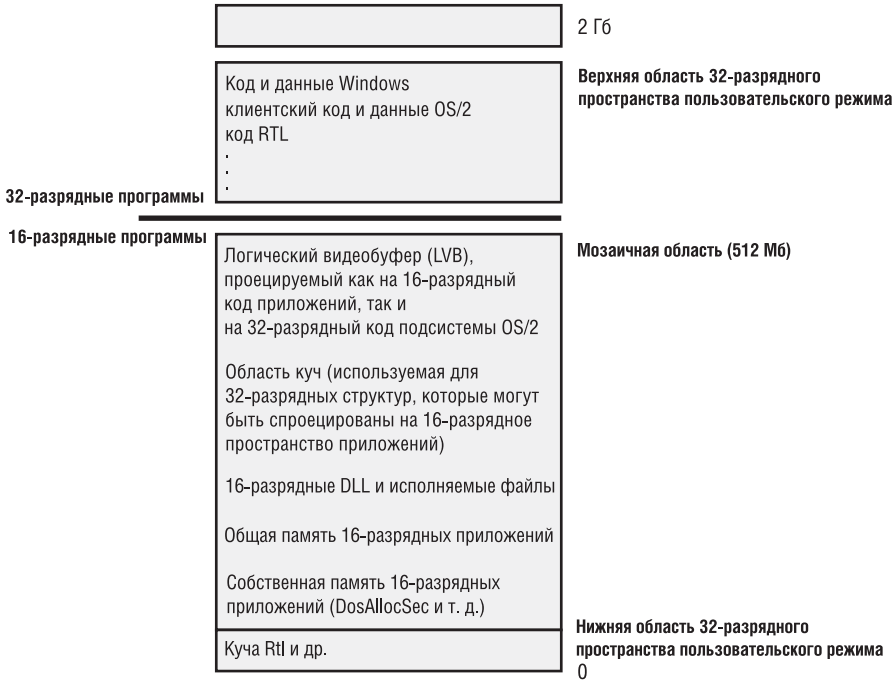


Рис. 2-5. *Структура виртуальной памяти подсистемы OS/2*

Как будет детально показано в главе 6, потоки являются элементами выполняемой программы и, как таковые, подлежат планированию (подключению к процессору по определенной схеме). В OS/2 всего 64 уровня приоритетов (от 0 до 63), а в Windows — 32 (от 0 до 31). Несмотря на это, 64 уровня приоритетов OS/2 проецируются на динамические приоритеты Windows с 1-го по 15-й. Потоки OS/2, выполняемые в Windows, никогда не получают приоритеты реального времени (16–31).

Как и подсистема POSIX, подсистема OS/2 автоматически запускается при первой активизации OS/2-совместимого образа и продолжает выполняться до перезагрузки всей системы.

Подробнее о выполнении приложений POSIX и OS/2 в Windows см. главу 6.

Ntdll.dll

Ntdll.dll — специальная библиотека системной поддержки, нужная в основном при использовании DLL подсистем. Она содержит функции двух типов:

- интерфейсы диспетчера системных сервисов (system service dispatch stubs) к сервисам исполнительной системы Windows;
- внутренние функции поддержки, используемые подсистемами, DLL подсистем и другими компонентами операционной системы.

Первая группа функций предоставляет интерфейс к сервисам исполнительной системы Windows, которые можно вызывать из пользовательского режима. Таких функций более 200, например *NtCreateFile*, *NtSetEvent* и т. д. Как уже говорилось, большинство из них доступно через Windows API (однако некоторые из них предназначены только для применения внутри самой операционной системы).

Для каждой из этих функций в Ntdll существует точка входа с тем же именем. Код внутри функции содержит специфичную для конкретной аппаратной архитектуры команду перехода в режим ядра для вызова диспетчера системных сервисов (о нем рассказывается в главе 3), который после проверки некоторых параметров вызывает уже настоящий сервис режима ядра из Ntoskrnl.exe.

Ntdll также включает множество функций поддержки, например загрузчик образов (функции, имена которых начинаются с *Ldr*), диспетчер куч, функции для взаимодействия с процессом подсистемы Windows (функции, имена которых начинаются с *Csr*), а также универсальные процедуры библиотек периода выполнения (функции, имена которых начинаются с *Rtl*). Там же находится диспетчер APC (asynchronous procedure call) пользовательского режима и диспетчер исключений. (Подробнее об APC и исключениях см. главу 3.)

Исполнительная система

Исполнительная система (executive) находится на верхнем уровне Ntoskrnl.exe (ядро располагается на более низком уровне). В ее состав входят функции следующего типа.

- Экспортируемые функции, доступные для вызова из пользовательского режима. Эти функции называются *системными сервисами* и экспортируются через Ntdll. Большинство сервисов доступно через Windows API или API других подсистем окружения. Однако некоторые из них недоступны через документированные функции (примером могут служить LPC, функции запросов вроде *NtQueryInformationProcess*, специализированные функции типа *NtCreatePagingFile* и т. д.).
- Функции драйверов устройств, вызываемые через функцию *DeviceIoControl*. Последняя является универсальным интерфейсом от пользовательского режима к режиму ядра для вызова функций в драйверах устройств, не связанных с чтением или записью.
- Экспортируемые функции, доступные для вызова только из режима ядра и документированные в Windows DDK или Windows Installable File System (IFS) Kit (см. *www.microsoft.com/whdc/ddk/ifskit.default.mspx*).

- Экспортируемые функции, доступные для вызова только из режима ядра, но не описанные в Windows DDK или IFS Kit (например, функции, которые используются видеодрайвером, работающим на этапе загрузки, и чьи имена начинаются с *Inbv*).
- Функции, определенные как глобальные, но не экспортируемые символы. Включают внутренние функции поддержки, вызываемые в Ntoskrnl; их имена начинаются с *Iop* (функции поддержки диспетчера ввода-вывода) или с *Mi* (функции поддержки управления памятью).
- Внутренние функции в каком-либо модуле, не определенные как глобальные символы.

Исполнительная система состоит из следующих основных компонентов (каждый из них подробно рассматривается в последующих главах книги).

- *Диспетчер конфигурации* (см. главу 4), отвечающий за реализацию и управление системным реестром.
- *Диспетчер процессов и потоков* (см. главу 6), создающий и завершающий процессы и потоки. Низкоуровневая поддержка процессов и потоков реализована в ядре Windows, а исполнительная система дополняет эти низкоуровневые объекты своей семантикой и функциями.
- *Монитор состояния защиты* (см. главу 8), реализующий политики безопасности на локальном компьютере. Он охраняет ресурсы операционной системы, осуществляя аудит и контролируя доступ к объектам в период выполнения.
- *Диспетчер ввода-вывода* (см. главу 9), реализующий аппаратно-независимый ввод-вывод и отвечающий за пересылку ввода-вывода нужным драйверам устройств для дальнейшей обработки.
- *Диспетчер Plug and Play* (см. главу 9), определяющий, какие драйверы нужны для поддержки конкретного устройства, и загружающий их. Требования каждого устройства в аппаратных ресурсах определяются в процессе перечисления. В зависимости от требований каждого устройства диспетчер PnP распределяет такие ресурсы, как порты ввода-вывода, IRQ, каналы DMA и области памяти. Он также отвечает за посылку соответствующих уведомлений об изменениях в аппаратном обеспечении системы (при добавлении или удалении устройств).
- *Диспетчер электропитания* (см. главу 9), который координирует события, связанные с электропитанием, и генерирует уведомления системы управления электропитанием, посылаемые драйверам. Когда система не занята, диспетчер можно настроить на остановку процессора для снижения энергопотребления. Изменение энергопотребления отдельных устройств возлагается на их драйверы, но координируется диспетчером электропитания.
- *Подпрограммы WDM Windows Management Instrumentation* (см. главу 4), позволяющие драйверам публиковать информацию о своих рабочих характеристиках и конфигурации, а также получать команды от службы

WMI пользовательского режима. Потребители информации WMI могут находиться как на локальной машине, так и на любом компьютере в сети.

- *Диспетчер кэша* (см. главу 11), повышающий производительность файлового ввода-вывода за счет сохранения в основной памяти дисковых данных, к которым недавно было обращение (это также уменьшает число обращений к диску для записи, поскольку модифицированные данные предварительно накапливаются в памяти в течение определенного периода). Как вы увидите, диспетчер кэша выполняет эту задачу, используя поддержку проецируемых файлов со стороны диспетчера памяти.
- *Диспетчер памяти* (см. главу 7), реализующий виртуальную память — схему управления памятью, позволяющую выделять каждому процессу большое закрытое адресное пространство, объем которого может превышать доступную физическую память. Диспетчер памяти также обеспечивает низкоуровневую поддержку диспетчера кэша.
- *Средство логической предвыборки* (см. главу 7), ускоряющее запуск системы и процессов за счет оптимизации загрузки данных, к которым происходит обращение при запуске системы или процессов.

Кроме того, в состав исполнительной системы входят четыре основные группы функций поддержки, используемые вышеперечисленными компонентами. Примерно треть из них описана в DDK, поскольку драйверы тоже используют их. Вот что представляют собой четыре категории функций поддержки.

- *Диспетчер объектов* — создает, управляет и удаляет объекты и абстрактные типы данных исполнительной системы, используемые для представления таких ресурсов операционной системы, как процессы, потоки и различные синхронизирующие объекты. Подробнее о диспетчере объектов см. главу 3.
- *Механизм LPC* (см. главу 3) — передает сообщения между клиентским и серверным процессами на одном компьютере. LPC является гибкой, оптимизированной версией RPC (remote procedure call), стандартного механизма взаимодействия между клиентскими и серверными процессами через сеть.
- Большой набор стандартных библиотечных функций для обработки строк, арифметических операций, преобразования типов данных и обработки структур безопасности.
- *Подпрограммы поддержки исполнительной системы*, например для выделения системной памяти (пулов подкачиваемых и неподкачиваемых страниц), доступа к памяти со взаимоблокировкой, а также два специальных типа синхронизирующих объектов: ресурс (resource) и быстродействующий мьютекс (fast mutex).

Ядро

Ядро состоит из набора функций в Ntoskrnl.exe, предоставляющих фундаментальные механизмы (в том числе планирования потоков и синхронизации), которые используются компонентами исполнительной системы и низкоуровневыми аппаратно-зависимыми средствами поддержки (диспетчеризации прерываний и исключений), различными в каждой процессорной архитектуре. Код ядра написан в основном на C, а ассемблер использовали лишь для решения специфических задач, трудно реализуемых на C.

Как и функции поддержки исполнительной системы, упоминавшиеся в предыдущем разделе, часть функций ядра описана в DDK (см. функции, чьи имена начинаются с *Ke*), поскольку они необходимы для реализации драйверов устройств.

Объекты ядра

Ядро состоит из низкоуровневых, четко определенных и хорошо предсказуемых примитивов и механизмов операционной системы, позволяющих компонентам исполнительной системы более высокого уровня выполнять свои функции. Ядро отделено от остальной части исполнительной системы; оно реализует системные механизмы и не участвует в принятии решений, связанных с системной политикой. Практически все такие решения, кроме планирования и диспетчеризации потоков, принимаются исполнительной системой.

Вне ядра исполнительная система представляет потоки и другие разделяемые ресурсы в виде объектов. Управление этими объектами требует определенных издержек, так как нужны описатели, позволяющие манипулировать объектами, средства защиты и квоты ресурсов, резервируемых при их создании. В ядре можно избежать таких издержек, поскольку оно реализует набор более простых объектов, называемых *объектами ядра* (kernel objects). Эти объекты позволяют ядру контролировать обработку данных процессором и поддерживают объекты исполнительной системы. Большинство объектов уровня исполнительной системы инкапсулирует один или более объектов ядра, включая в себя их атрибуты, определенные ядром.

Одна из групп объектов ядра, называемых *управляющими* (control objects), определяет семантику управления различными функциями операционной системы. В эту группу входят объекты APC, DPC (deferred procedure call) и несколько объектов, используемых диспетчером ввода-вывода (например, объект прерывания).

Другая группа объектов под названием *объекты диспетчера* (dispatcher objects) реализует средства синхронизации, позволяющие изменять планирование потоков. В группу таких объектов входят поток ядра (kernel thread), мьютекс (mutex), событие (event), семафор (semaphore), таймер (timer), ожидаемый таймер (waitable timer) и некоторые другие. С помощью функций ядра исполнительная система создает объекты ядра, манипулирует ими и конструирует более сложные объекты, предоставляемые в пользовательском

режиме. Объекты подробно рассматриваются в главе 3, а процессы и потоки — в главе 6.

Поддержка оборудования

Другая важная задача ядра — абстрагирование или изоляция исполнительной системы и драйверов устройств от различий между аппаратными архитектурами, поддерживаемыми Windows (т. е. различий в обработке прерываний, диспетчеризации исключений и синхронизации между несколькими процессорами).

Архитектура ядра нацелена на максимальное обобщение кода — даже в случае аппаратно-зависимых функций. Ядро поддерживает набор семантически идентичных и переносимых между архитектурами интерфейсов. Большая часть кода, реализующего переносимые интерфейсы, также идентична для разных архитектур.

Но одна часть этих интерфейсов по-разному реализуется на разных архитектурах, а другая включает код, специфичный для конкретной архитектуры. Архитектурно-независимые интерфейсы могут быть вызваны на любой машине, причем семантика интерфейса будет одинаковой — независимо от специфического кода для той или иной архитектуры. Некоторые интерфейсы ядра (например, процедуры спин-блокировки, описанные в главе 3) на самом деле реализуются в HAL (см. следующий раздел), поскольку их реализация может отличаться даже в пределах семейства процессоров с одинаковой архитектурой.

В ядре также содержится небольшая порция кода с x86-специфичными интерфейсами, необходимыми для поддержки старых программ MS-DOS. Эти интерфейсы не являются переносимыми в том смысле, что их нельзя вызывать на машине с другой архитектурой, где они попросту отсутствуют. Этот x86-специфичный код, например, поддерживает манипуляции с GDT (global descriptor tables) и LDT (local descriptor tables) — аппаратными средствами x86.

Другим примером архитектурно-специфичного кода ядра может служить интерфейс, предоставляющий поддержку буфера трансляции и процессорного кэша. Эта поддержка требует разного кода на разных архитектурах, поскольку кэш в них реализуется различным образом.

Еще один пример — переключение контекста. Хотя на высоком уровне для выбора потоков и переключения контекста применяется один и тот же алгоритм (сохраняется контекст предыдущего потока, загружается контекст нового и запускается новый поток), существуют архитектурные различия между его реализациями для разных процессоров. Поскольку контекст описывается состоянием процессора (его регистров и т. д.), сохраняемая и загружаемая информация зависит от архитектуры.

Уровень абстрагирования от оборудования

Как отмечалось в начале этой главы, одной из важнейших особенностей архитектуры Windows является переносимость между различными аппаратными

платформами. Ключевой компонент, обеспечивающий такую переносимость, — уровень абстрагирования от оборудования (hardware abstraction layer, HAL). HAL — это загружаемый модуль режима ядра (Hal.dll), предоставляющий низкоуровневый интерфейс с аппаратной платформой, на которой выполняется Windows. Он скрывает от операционной системы специфику конкретной аппаратной платформы, в том числе ее интерфейсов ввода-вывода, контроллеров прерываний и механизмов взаимодействия между процессорами, т. е. все функции, зависимые от архитектуры и от конкретной машины.

Когда внутренним компонентам Windows и драйверам устройств нужна платформенно-зависимая информация, они обращаются не к самому оборудованию, а к подпрограммам HAL, что и обеспечивает переносимость этой операционной системы. По этой причине подпрограммы HAL документированы в Windows DDK, где вы найдете более подробные сведения о HAL и о его использовании драйверами.

Хотя в Windows имеется несколько модулей HAL (см. таблицу 2-6), при установке на жесткий диск компьютера копируется только один из них — Hal.dll. (В других операционных системах, например в VMS, нужный модуль HAL выбирается при загрузке системы.) Поскольку для поддержки разных процессоров требуются разные модули HAL, системный диск от одной x86-установки скорее всего не подойдет для загрузки системы с другим процессором.

Таблица 2-6. *Список модулей HAL для x86 в \Windows\Driver\Cache\i386\Driver.cab*

Имя файла HAL	Поддерживаемые системы
Hal.dll	Стандартные персональные компьютеры (ПК)
Halacpi.dll	ПК с ACPI (Advanced Configuration and Power Interface)
Halapic.dll	ПК с APIC (Advanced Programmable Interrupt Controller)
Halaacpi.dll	ПК с APIC и ACPI
Halmps.dll	Многопроцессорные ПК
Halmacpi.dll	Многопроцессорные ПК с ACPI
Halborg.dll	Рабочие станции Silicon Graphics (только для Windows 2000; больше не продаются)
Halsp.dll	Compaq SystemPro (только для Windows XP)

ПРИМЕЧАНИЕ В базовой системе Windows Server 2003 нет HAL, специфических для конкретных вендоров.

ЭКСПЕРИМЕНТ: просмотр базовых HAL, включенных в Windows

Для просмотра HAL, включенных в Windows, откройте файл Driver.cab в соответствующем подкаталоге, специфичном для конкретной архитектуры, в каталоге \Windows\Driver Cache. (Например, для систем x86 имя этого файла — \Windows\Driver Cache\i386\Driver.cab.) Прокрутите список до файлов, начинающихся с «Hal», и вы увидите файлы, перечисленные в таблице 2-6.

ЭКСПЕРИМЕНТ: определяем используемый модуль HAL

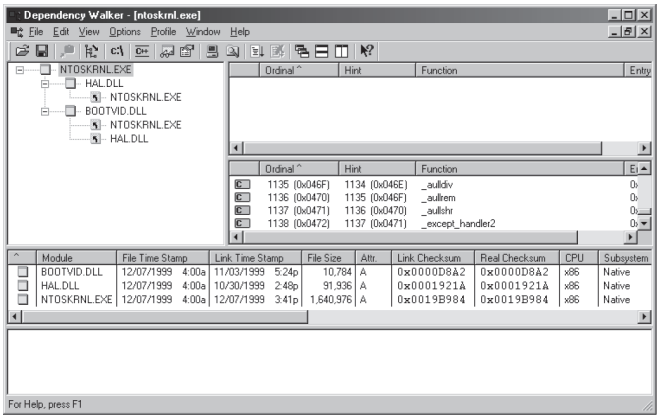
Определить, какой модуль HAL используется на вашей машине, можно двумя способами.

1. Откройте файл \Windows\Repair\Setup.log, найдите строку с *Hal.dll*. Имя файла, стоящее в этой строке после знака равенства, соответствует имени модуля HAL, извлеченного из Driver.cab с дистрибутивного носителя.
2. Откройте Device Manager (Диспетчер устройств): щелкните правой кнопкой мыши значок My Computer (Мой компьютер) на рабочем столе, выберите команду Properties (Свойства), откройте вкладку Hardware (Оборудование) и щелкните кнопку Device Manager (Диспетчер устройств). Проверьте имя «драйвера» для устройства Computer (Компьютер).

ЭКСПЕРИМЕНТ: просмотр зависимостей NTOSKRNL и HAL

Вы можете просмотреть взаимосвязи образов ядра и HAL, изучив их таблицы импорта и экспорта с помощью утилиты Dependency Walker (Depends.exe), которая содержится в Windows Support Tools и Platform SDK. Для исследования файла в Dependency Walker откройте его командой Open из меню File.

Вот пример вывода этой утилиты при просмотре зависимостей в Ntoskrnl.



Обратите внимание, что Ntoskrnl связан с HAL, который в свою очередь связан с Ntoskrnl (оба используют функции друг у друга). Ntoskrnl также связан с Bootvid.dll, видеодрайвером, используемым для вывода заставки при запуске Windows. В Windows XP и выше вы увидите в списке дополнительную DLL, Kdcom.dll. Она содержит код инфраструктуры отладчика ядра, который раньше был частью Ntoskrnl.exe.

Подробное описание информации, выводимой Dependency Walker, см. в справочном файле этой утилиты (Depends.hlp).

Драйверы устройств

Драйверы устройств подробно описываются в главе 9, а здесь мы даем краткий обзор их типов и поясняем, как перечислить установленные драйверы, загруженные в системе.

Драйверы устройств являются загружаемыми модулями режима ядра (как правило, это файлы с расширением .sys); они образуют интерфейс между диспетчером ввода-вывода и соответствующим оборудованием. Эти драйверы выполняются в режиме ядра в одном из трех контекстов:

- в контексте пользовательского потока, инициировавшего функцию ввода-вывода;
- в контексте системного потока режима ядра;
- как результат прерывания (а значит, не в контексте какого-либо процесса или потока, который был текущим на момент прерывания).

Как было сказано в предыдущем разделе, в Windows драйверы устройств не управляют оборудованием напрямую — вместо этого они вызывают функции HAL. Драйверы, как правило, пишутся на C (иногда на C++), поэтому при правильном использовании процедур HAL они являются переносимыми между поддерживаемыми Windows архитектурами на уровне исходного кода, а на уровне двоичных файлов — внутри семейства с одинаковой архитектурой. Существует несколько типов драйверов устройств.

- *Драйверы аппаратных устройств*, которые управляют (через HAL) оборудованием, записывают на них выводимые данные и получают вводимые данные от физического устройства или из сети. Есть множество типов таких драйверов — драйверы шин, интерфейсов, устройств массовой памяти и т. д.
- *Драйверы файловой системы* — это драйверы Windows, принимающие запросы на файловый ввод-вывод и транслирующие их в запросы ввода-вывода для конкретного устройства.
- *Драйверы фильтра файловой системы*, которые обеспечивают зеркалирование и шифрование дисков, перехват ввода-вывода и некоторую дополнительную обработку информации перед передачей ее на следующий уровень.
- *Сетевые редиректоры* и *серверы*, являющиеся драйверами файловых систем, которые передают запросы файловой системы на ввод-вывод другим компьютерам в сети и принимают от них аналогичные запросы.
- *Драйверы протоколов*, реализующие сетевые протоколы вроде TCP/IP, NetBEUI и IPX/SPX.
- *Драйверы потоковых фильтров ядра*, действующие по цепочке для обработки потоковых данных, например при записи и воспроизведении аудио- и видеоинформации.

Поскольку установка драйвера устройства — единственный способ добавления в систему стороннего кода режима ядра, некоторые программисты

пишут драйверы просто для того, чтобы получить доступ к внутренним функциям или структурам данных операционной системы, недоступным из пользовательского режима (но документированным и поддерживаемым в DDK). Например, многие утилиты с *www.sysinternals.com* представляют собой комбинацию GUI-приложений Windows с драйверами устройств, используемыми для сбора сведений о состоянии внутрисистемных структур и вызова функций, доступных только в режиме ядра.

Усовершенствования в модели драйверов Windows

В Windows 2000 была введена поддержка Plug and Play и энергосберегающих технологий, а также расширена модель драйверов Windows NT, называемая Windows Driver Model (WDM). Windows 2000 и более поздние версии могут работать с унаследованными драйверами Windows NT 4, но, поскольку они не поддерживают Plug and Play и энергосберегающие технологии, функциональность системы в этом случае будет ограничена.

С точки зрения WDM, существует три типа драйверов.

- *Драйвер шины* (bus driver), обслуживающий контроллер шины, адаптер, мост или любые другие устройства, имеющие дочерние устройства. Драйверы шин нужны для работы системы и в общем случае поставляются Microsoft. Для каждого типа шины (PCI, PCMCIA и USB) в системе имеется свой драйвер. Сторонние разработчики создают драйверы для поддержки новых шин вроде VMEbus, Multibus и Futurebus.
- *Функциональный драйвер* (function driver) — основной драйвер устройства, предоставляющий его функциональный интерфейс. Обязателен, кроме тех случаев, когда устройство используется без драйверов (т. е. ввод-вывод осуществляется драйвером шины или драйверами фильтров шины, как в случае SCSI PassThru). Функциональный драйвер по определению обладает наиболее полной информацией о своем устройстве. Обычно только этот драйвер имеет доступ к специфическим регистрам устройства.
- *Драйвер фильтра* (filter driver), поддерживающий дополнительную функциональность устройства (или существующего драйвера) или изменяющий запросы на ввод-вывод и ответы на них от других драйверов (это часто используется для коррекции устройств, предоставляющих неверную информацию о своих требованиях к аппаратным ресурсам). Такие драйверы не обязательны, и их может быть несколько. Они могут работать как на более высоком уровне, чем функциональный драйвер или драйвер шины, так и на более низком. Обычно эти драйверы предоставляются OEM-производителями или независимыми поставщиками оборудования (IHV).

В среде WDM один драйвер не может контролировать все аспекты устройства: драйвер шины информирует диспетчер PnP об устройствах, подключенных к шине, в то время как функциональный драйвер управляет устройством.

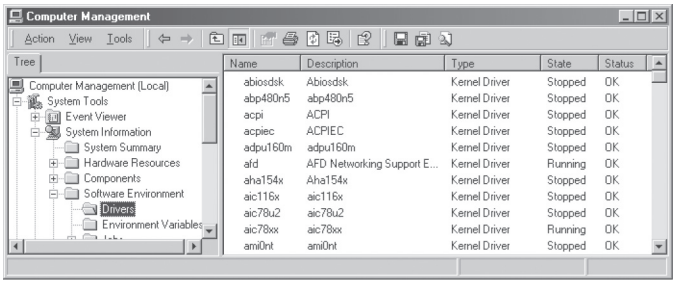
В большинстве случаев драйвер фильтра более низкого уровня модифицирует поведение устройства. Например, если устройство сообщает драйверу своей шины о том, что ему нужно 4 порта ввода-вывода, тогда как на самом деле ему требуется 16, драйвер фильтра может перехватить список аппаратных ресурсов, направляемый драйвером шины диспетчеру PnP и исправить число портов.

Драйвер фильтра более высокого уровня обычно придает устройству дополнительную функциональность. Так, высокоуровневый драйвер фильтра для клавиатуры может обеспечивать дополнительную защиту.

Обработка прерываний описывается в главе 3. Подробнее о диспетчере ввода-вывода, WDM, Plug and Play и энергосберегающих технологиях см. главу 9.

ЭКСПЕРИМЕНТ: просмотр установленных драйверов устройств

Чтобы вывести список установленных драйверов, запустите оснастку Computer Management (Управление компьютером). Для этого выберите из меню Start (Пуск) команду Programs (Программы), затем Administrative Tools (Администрирование) и Computer Management (Управление компьютером) или откройте Control Panel (Панель управления) и дважды щелкните значок Computer Management. В окне Computer Management раскройте System Information (Сведения о системе), затем Software Environment (Программная среда) и Drivers (Драйверы). Ниже приведен пример списка драйверов.



В этом окне выводится список драйверов, определенных в реестре, а также их тип и состояние — Running (Работает) или Stopped (Остановлена). Драйверы устройств и процессы Windows-сервисов определяются в разделе реестра HKLM\SYSTEM\CurrentControlSet\Services. Однако они отличаются по коду типа, например 1 соответствует драйверу режима ядра. (Полный список хранящихся в реестре сведений, которые относятся к драйверам, см. в таблице 4-7.)

Список загруженных в текущий момент драйверов можно просмотреть и с помощью утилиты Drivers (Drivers.exe в ресурсах Windows 2000) или Pstat (Pstat.exe в Windows XP Support Tools, Windows Server 2003 Support Tools, ресурсах Windows 2000 и Platform SDK). Ниже приведен листинг части выходной информации утилиты Drivers.

см. след. стр.

```
C:\>drivers
  ModuleName   Code   Data  Bss   Paged    Init          LinkDate
---------------------------------
ntoskrnl.exe 429184  96896   0  775360  138880  Tue Dec 07 18:41:11 1999
     hal.dll  25856   6016   0   16160   10240  Tue Nov 02 20:14:22 1999
 BOOTVID.DLL   5664   2464   0       0     320  Wed Nov 03 20:24:33 1999
    ACPI.sys  92096   8960   0   43488    4448  Wed Nov 10 20:06:04 1999
  WMILIB.SYS    512      0   0    1152     192  Sat Sep 25 14:36:47 1999
     pci.sys  12704   1536   0   31264    4608  Wed Oct 27 19:11:08 1999
  isapnp.sys  14368    832   0   22944    2048  Sat Oct 02 16:00:35 1999
compbatt.sys   2496      0   0    2880    1216  Fri Oct 22 18:32:49 1999
   BATTC.SYS    800      0   0    2976     704  Sun Oct 10 19:45:37 1999
intelide.sys   1760     32   0       0     128  Thu Oct 28 19:20:03 1999
 PCIIDEX.SYS   4544    480   0   10944    1632  Wed Oct 27 19:02:19 1999
  pcmcia.sys  32800   8864   0   23680    6240  Fri Oct 29 19:20:08 1999
  ftdisk.sys   4640     32   0   95072    3392  Mon Nov 22 14:36:23 1999
---------------------------------
      Total 4363360  580320  0 3251424  432992
```

Утилита перечисляет все загруженные компоненты режима ядра (Ntoskrnl, HAL и драйверы устройств) и сообщает размеры разделов в каждом образе.

Pstat также выводит список загруженных драйверов, но только после списка процессов и потоков в каждом процессе. Она показывает один вид очень важной информации, не сообщаемой утилитой Drivers: адрес загрузки модуля в системном пространстве. Как вы еще увидите, этот адрес нужен для увязки выполняемых системных потоков с драйвером, в котором они существуют.

Недокументированные интерфейсы

Просмотр имен экспортируемых или глобальных символов в ключевых системных файлах (Ntoskrnl.exe, Hal.dll или Ntdll.dll) может оказаться полезным: вы получите представление о том, что умеет делать Windows в сравнении с документированной и поддерживаемой частью. Конечно, знание имен этих функций еще не означает, что вы сможете или должны их вызывать. Эти интерфейсы не документированы и могут быть изменены. Мы предлагаем рассмотреть эти функции только для лучшего понимания внутренних функций Windows, а не для обхода поддерживаемых интерфейсов.

Например, просмотрев список функций в Ntdll.dll, вы сможете сравнить список всех системных сервисов, которые Windows предоставляет DLL-модулям подсистем пользовательского режима, с их подмножеством, предоставляемым каждой подсистемой. Хотя многие из этих функций точно соответствуют документированным и поддерживаемым Windows-функциям, некоторые из них недоступны через Windows API (см. статью «Inside the Native API» на *www.sysinternals.com*).

С другой стороны, было бы интересно выяснить, что импортируют DLL-модули подсистемы Windows (скажем, Kernel32.dll или Advapi32.dll) и какие функции они вызывают в Ntdll.

Также представляет интерес содержимое Ntoskrnl.exe: хотя в Windows DDK документированы многие экспортируемые подпрограммы, используемые драйверами режима ядра, немалая их часть не описана. Возможно, вас заинтересует содержимое таблицы импорта для Ntoskrnl и HAL, где перечислены функции HAL, используемые Ntoskrnl, и наоборот.

В таблице 2-7 приведено большинство общеупотребительных префиксов имен функций в компонентах исполнительной системы. В каждом из таких компонентов также используются слегка модифицированные префиксы, обозначающие внутренние функции: либо за первой буквой префикса указывается *i* (от *internal*), либо префикс заканчивается на букву *p* (от *private*). Так, *Ki* относится к внутренним функциям ядра, а *Psp* — к внутренним функциям поддержки процессов.

Таблица 2-7. *Общеупотребительные префиксы*

Префикс	Компонент
Cc	Диспетчер кэша
Cm	Диспетчер конфигурации
Ex	Подпрограммы поддержки исполнительной системы
FsRtl	Библиотека (периода выполнения) драйвера файловой системы
Hal	HAL
Io	Диспетчер ввода-вывода
Ke	Ядро
Lpc	LPC
Lsa	Локальная аутентификация
Mm	Диспетчер памяти
Nt	Системные сервисы Windows (большинство из них экспортируется как Windows-функции)
Ob	Диспетчер объектов
Po	Диспетчер электропитания
Pp	Диспетчер PnP
Ps	Поддержка процессов
Rtl	Стандартная библиотека (периода выполнения)
Se	Защита
Wmi	Windows Management Instrumentation (Инструментарий управления Windows)
Zw	Зеркальная точка входа для системных сервисов (имена которых начинаются с *Nt*), устанавливающих предыдущий режим доступа к ядру. При этом параметры не контролируются, так как *Nt*-сервисы проверяют параметры только при переключении из пользовательского режима

см. след. стр.

Понимая схему именования системных процедур Windows, расшифровывать имена экспортируемых функций гораздо легче. Общий формат выглядит так:

```
<префикс><операция><объект>
```

где *префикс* — внутренний компонент, экспортирующий процедуру, *операция* — название операции, выполняемой над объектом или ресурсом, а *объект* — объект, над которым проводится эта операция.

Например, *ExAllocatePoolWithTag* является процедурой поддержки исполнительной системы, которая выделяет память из пула подкачиваемых или неподкачиваемых страниц. *KeInitializeThread* представляет собой процедуру для создания и инициализации объекта ядра «поток».

Системные процессы

В каждой системе Windows выполняются перечисленные ниже процессы. (Два из них, Idle и System, не являются процессами в строгом смысле этого слова, поскольку они не выполняют какой-либо код пользовательского режима.)

- Процесс Idle (включает по одному потоку на процессор для учета времени простоя процессора).
- Процесс System (содержит большинство системных потоков режима ядра).
- Диспетчер сеансов (Smss.exe).
- Подсистема Windows (Csrss.exe).
- Процесс входа в систему (Winlogon.exe).
- Диспетчер управления сервисами (Services.exe) и создаваемые им дочерние процессы сервисов (например, универсальный процесс для хостинга сервисов, Svchost.exe).
- Серверный процесс локальной аутентификации (Lsass.exe).

Чтобы понять взаимоотношения этих процессов, полезно просмотреть «дерево» процессов, отражающее связи между родительскими и дочерними процессами. Увидев, кем создается тот или иной процесс, вам будет легче понять, откуда берется каждый процесс. На рис. 2-6 показана часть экранного снимка дерева процессов с комментариями по нескольким первым процессам. (Process Explorer позволяет добавлять комментарии для индивидуальных процессов и выводить их как дополнительную колонку в окне.)

В следующих разделах поясняются основные системные процессы, перечисленные на рис. 2-6. Хотя в этих разделах дается краткое описание последовательности запуска данных процессов, подробно все этапы загрузки Windows рассматриваются в главе 5.

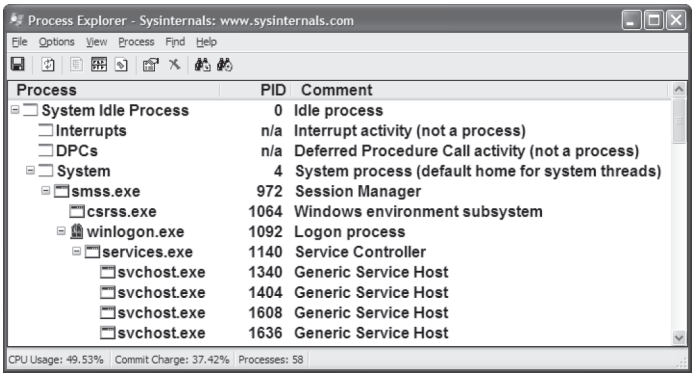


Рис. 2-6. *Начальное дерево системных процессов*

Процесс Idle

Первый процесс, показанный на рис. 2-6, является процессом простоя системы (system idle process). Как будет показано в главе 6, процессы идентифицируются по именам их образов. Однако этот процесс (как и процесс System) не выполняет реальный код пользовательского режима (в том смысле, что в каталоге \Windows нет «System Idle Process.exe»). Кроме того, разные утилиты из-за особенностей реализации по-разному именуют его. В таблице 2-8 приводится несколько имен процесса Idle (с идентификатором 0); подробнее о нем рассказывается в главе 6.

Таблица 2-8. *Имена процесса с идентификатором 0, сообщаемые различными утилитами*

Утилита	Имя процесса с идентификатором 0
Task Manager	System Idle Process
Process Viewer (Pviewer.exe)	Idle
Process Status (Pstat.exe)	Idle Process
Process Explode (Pview.exe)	System Process
Task List (Tlist.exe)	System Process
QuickSlice (Qslice.exe)	Systemprocess

А теперь рассмотрим системные потоки и предназначение каждого системного процесса, выполняющего реальный код.

Прерывания и DPC

Две строки, помеченные как Interrupts и DPCs, отражают время, затраченное на обслуживание прерываний и обработку отложенных вызовов процедур (deferred procedure calls, DPC). Эти механизмы объясняются в главе 3. Заметьте: хотя Process Explorer показывает эти строки в списке процессов, они не имеют отношения к процессам. Они выводятся потому, что ведут учет процессорного времени, не выделенного какому-либо процессу. Но Task Manager (Диспетчер задач) рассматривает время, затраченное на обработку преры-

ваний и DPC, как время простоя системы. Поэтому система, занятая интенсивной обработкой прерываний, будет выглядеть в Task Manager так, будто она ничем не занимается.

Процесс System и его потоки

Процесс System (с идентификатором 8 в Windows 2000 и идентификатором 4 в Windows XP и Windows Server 2003) служит носителем особых потоков, работающих только в режиме ядра, — *системных потоков режима ядра* (kernel-mode system threads). У системных потоков имеются все атрибуты и контексты обычных потоков пользовательского режима (например, контекст оборудования, приоритет и т. д.), но они отличаются тем, что выполняются только в режиме ядра внутри системного кода, загруженного в системное пространство, — будь то Ntoskrnl.exe или какой-либо драйвер устройства. Кроме того, у системных потоков нет адресного пространства пользовательского процесса, и поэтому нужная им динамическая память выделяется из куч памяти операционной системы, например из пула подкачиваемых или неподкачиваемых страниц.

Системные потоки создаются функцией *PsCreateSystemThread* (документирована в DDK), вызываемой только в режиме ядра. Windows, как и драйверы устройств, создает системные потоки при инициализации системы для выполнения действий, требующих получения контекста потока, например для выдачи и ожидания запросов на ввод-вывод или опроса устройства. Скажем, диспетчер памяти использует системные потоки для реализации таких функций, как запись измененных страниц в страничный файл (page file) или в спроецированные файлы, загрузки процессов в память или выгрузки из нее и т. д. Ядро создает системный поток под названием «диспетчер настройки баланса» (balance set manager), активизируемый раз в секунду для инициации при необходимости различных событий, связанных с планированием и управлением памятью. Диспетчер кэша также использует системные потоки для реализации как опережающего чтения, так и отложенной записи. Драйвер файл-сервера (Srv.sys) с помощью системных потоков отвечает на сетевые запросы ввода-вывода применительно к файлам на общих дисковых разделах, доступных в сети. Даже драйвер дисковода гибких дисков создает свой системный поток для опроса этого устройства (это повышает эффективность опроса, потому что драйвер дисковода гибких дисков, управляемый прерываниями, расходует много системных ресурсов). Подробнее о конкретных системных потоках см. главы, где рассматриваются соответствующие компоненты.

По умолчанию владельцем системных потоков является процесс System, но драйверы могут создавать системные потоки в любом процессе. Например, драйвер подсистемы Windows (Win32k.sys) создает системные потоки в процессе подсистемы Windows (Csrss.exe), чтобы облегчить доступ к данным в адресном пространстве этого процесса в пользовательском режиме.

Если вы занимаетесь поиском причин неполадок или системным анализом, полезно сопоставить выполнение индивидуальных системных потоков

с создавшими их драйверами или даже с подпрограммой, содержащей соответствующий код. Так, на сильно загруженном файл-сервере процесс System скорее всего потребляет значительную часть процессорного времени. Но для определения того, какой именно драйвер или компонент операционной системы выполняется, просто знать, что процесс System в данный момент выполняет «какой-то системный поток», недостаточно.

Так что, если в процессе System выполняются потоки, сначала определите, какие это потоки (например, с помощью оснастки Performance). Найдя такой поток (или потоки), посмотрите, в каком драйвере началось выполнение системного потока (это по крайней мере укажет на наиболее вероятного создателя потока), либо проанализируйте стек вызовов (или хотя бы текущий адрес) интересующего вас потока, что позволит определить, в каком месте он сейчас выполняется.

Оба этих метода демонстрируются следующими экспериментами.

ЭКСПЕРИМЕНТ: идентификация системных потоков в процессе System

Вы можете убедиться, что потоки внутри процесса System должны быть потоками режима ядра, поскольку стартовый адрес каждого из них больше адреса начала системного пространства (которое по умолчанию начинается с 0x80000000, если система загружена без параметра /3GB в Boot.ini). Кроме того, обратив внимание на процессорное время, выделяемое этим потокам, вы увидите, что они занимают процессорное время только при выполнении в режиме ядра. Чтобы определить драйвер, создавший системный поток, найдите стартовый адрес потока (с помощью Pviewer.exe) и ищите драйвер с базовым адресом, ближайшим (с меньшей стороны) к этому стартовому адресу. Утилита Pstat в конце своих выходных данных, как и команда *!drivers* отладчика ядра, сообщает базовые адреса каждого загруженного драйвера устройства.

Чтобы быстро найти текущий адрес потока, воспользуйтесь командой *!stacks 0* отладчика ядра. Ниже приводится образец вывода, полученный на работающей системе с помощью LiveKd.

```
kd> !stacks 0
Proc.Thread  Thread  ThreadState  Blocker
                                  [System]
 8.000004  8146edb0  BLOCKED  ntoskrnl!MmZeroPageThread+0x5f
 8.00000c  8146e730  BLOCKED  ?? Kernel stack not resident ??
 8.000010  8146e4b0  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.000014  8146d030  BLOCKED  ?? Kernel stack not resident ??
 8.000018  8146ddb0  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.00001c  8146db30  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.000020  8146d8b0  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.000024  8146d630  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.000028  8146d3b0  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.00002c  8146c030  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
```

см. след. стр.

```
8.000030  8146cdb0  BLOCKED  ntoskrnl!ExpWorkerThreadBalanceManager+0x55
8.000034  8146b470  BLOCKED  ntoskrnl!MiDereferenceSegmentThread+0x44
8.000038  8146b1f0  BLOCKED  ntoskrnl!MiModifiedPageWriterWorker+0x31
8.00003c  8146a030  BLOCKED  ntoskrnl!KeBalanceSetManager+0x7e
8.000040  8146adb0  BLOCKED  ntoskrnl!KeSwapProcessOrStack+0x24
8.000044  8146a5b0  BLOCKED  ntoskrnl!FsRtlWorkerThread+0x33
8.000048  8146a330  BLOCKED  ntoskrnl!FsRtlWorkerThread+0x33
8.00004c  81461030  BLOCKED  ACPI!ACPIWorker+0x46
8.000050  8143a770  BLOCKED  ntoskrnl!MiMappedPageWriter+0x4d
8.000054  81439730  BLOCKED  dmio!voliod_loop+0x399
8.000058  81436c90  BLOCKED  NDIS!ndisWorkerThread+0x22
8.00005c  813d9170  BLOCKED  ltmdmntt!WakeupTimerThread+0x27
8.000060  813d8030  BLOCKED  ltmdmntt!WriteRegistryThread+0x1c
8.000070  8139c850  BLOCKED  raspptp!MainPassiveLevelThread+0x78
8.000074  8139c5d0  BLOCKED  raspptp!PacketWorkingThread+0xc0
8.00006c  81384030  BLOCKED  rasacd!AcdNotificationRequestThread+0xd8
8.000080  81333330  BLOCKED  rdbss!RxpWorkerThreadDispatcher+0x6f
8.000084  813330b0  BLOCKED  rdbss!RxSpinUpRequestsDispatcher+0x58
8.00008c  81321db0  BLOCKED  ?? Kernel stack not resident ??
8.00015c  81205570  BLOCKED  INO_FLTR+0x68bd
8.000160  81204570  BLOCKED  INO_FLTR+0x80e9
8.000178  811fcdb0  BLOCKED  irda!RxThread+0xfa
8.0002d0  811694f0  BLOCKED  ?? Kernel stack not resident ??
8.0002d4  81168030  BLOCKED  ?? Kernel stack not resident ??
8.000404  811002b0  BLOCKED  rdbss!RxpWorkerThreadDispatcher+0x6f
8.000430  810f4990  READY    parallel!ParallelThread+0x3e
8.00069c  80993030  READY    rdbss!RxpWorkerThreadDispatcher+0x6f
```

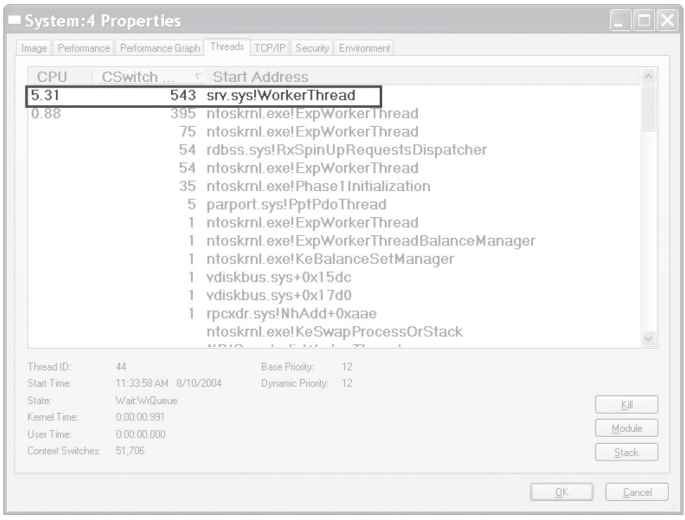
В первом столбце выводятся идентификаторы процесса и потока (в виде «Идентификатор процесса. Идентификатор потока»). Во втором сообщается текущий адрес потока, в третьем — состояние потока: ожидает, готов или выполняется (о состояниях потоков см. главу 6). В последнем столбце показывается адрес вершины стека потока. Эта информация помогает определить драйвер, в котором началось выполнение того или иного потока. Имя функции потока в Ntoskrnl дает дополнительную подсказку о том, что именно делает поток.

Однако, если выполняемый поток является одним из рабочих потоков системы (ExpWorkerThread), вы не сможете точно сказать, что он делает, поскольку любой драйвер может давать задания рабочему потоку системы. В этом случае единственный выход — поставить точку прерывания в *ExQueueWorkItem*. По достижении этой точки введите **!dso work_queue_item esp+4**. Эта команда даст вам первый аргумент функции *ExQueueWorkItem*, представляющий собой указатель на структуру рабочего элемента, которая в свою очередь содержит адрес процедуры рабочего потока, вызываемой в его контексте. В качестве альтернативы можно использовать команду **k** отладчика ядра, которая сообщает текущее содержимое стека вызовов. А это подскажет, какой драйвер отправил задание рабочему потоку.

ЭКСПЕРИМЕНТ: увязка системного потока с драйвером устройства

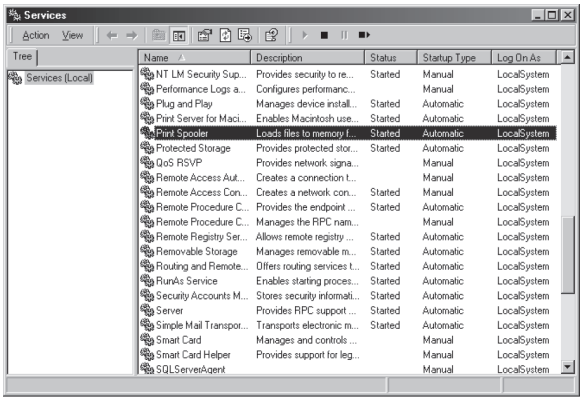
В этом эксперименте мы посмотрим, как увязать активность процессора в процессе System с системным потоком (и драйвером, к которому он относится), вызывающим эту активность. Это важно: чтобы по-настоящему понять, что происходит, нужно перейти на уровень потоков процесса System. В данном случае мы вызовем активность системного потока, создав нагрузку файлового сервера на компьютере. (Драйвер файл-сервера Srv.sys создает системные потоки для обработки входящих запросов на файловый ввод-вывод. Подробнее об этом компоненте см. главу 13.)

1. Откройте окно командной строки.
2. Создайте список всех каталогов на диске C, используя сетевой путь для доступа к этому диску. Например, если имя вашего компьютера — COMPUTER1, введите dir **\\computer1\c$ /s**. (Ключ ***/s*** заставляет перечислять все подкаталоги.)
3. Запустите Process Explorer и дважды щелкните процесс System.
4. Откройте вкладку Threads.
5. Отсортируйте список по столбцу CSwitch Delta (разница по числу переключений контекста). Вы должны увидеть один или более потоков в Srv.sys, как показано на следующей иллюстрации.



Если вы видите работающий системный поток и не уверены, какой это драйвер, нажмите кнопку Module, которая открывает окно свойств для файла. Например, нажатие кнопки Module при выбранном, как на предыдущей иллюстрации, потоке в Srv.sys выводит результаты в следующем окне.

см. след. стр.



Диспетчер сеансов (Smss)

Диспетчер сеансов (Session Manager) (\Windows\System32\Smss.exe) является первым процессом пользовательского режима, создаваемым в системе. Он порождается системным потоком режима ядра, отвечающим за последний этап инициализации исполнительной системы и ядра.

Диспетчер сеансов отвечает за некоторые важные этапы запуска Windows, такие как создание дополнительных страничных файлов, выполнение отложенных операций по копированию, переименованию и удалению файлов, а также создание системных переменных окружения. Он также запускает процессы подсистем (обычно только Csrss.exe) и Winlogon, который в свою очередь создает остальные системные процессы.

Значительная часть сведений о конфигурации, хранящихся в реестре и используемых при инициализации Smss, находится в HKLM\SYSTEM\CurrentControlSet\Control\Session Manager. Некоторые из этих данных поясняются в главе 5 в разделе по Smss. (Более подробное описание разделов и параметров см. в справочном файле Regentry.chm из ресурсов Windows 2000).

После выполнения этих этапов инициализации главный поток Smss переходит к бесконечному ожиданию описателей процессов Csrss и Winlogon. Так как от них зависит функционирование Windows, при неожиданном завершении любого из них Smss вызывает крах системы (с кодом STATUS_SYSTEM_ PROCESS_TERMINATED, или 0xC000021A). Smss также ожидает запросы на загрузку, события отладки и запросы на запуск новых сеансов сервера терминала. (Описание служб терминала см. в главе 1.)

Сеанс Terminal Services создается Smss. Когда Smss получает запрос на создание сеанса, он сначала вызывает *NtSetSystemInformation* с запросом на настройку сеансовых структур данных режима ядра. Это приводит к вызову внутренней функции диспетчера памяти *MmSessionCreate*, настраивающей виртуальное адресное пространство сеанса, которое будет содержать пул подкачиваемой памяти для сеанса и сеансовые структуры данных, создаваемые подсистемой Windows (а точнее, ее частью, работающей в режиме

ядра), а также другими драйверами устройств. (Детали см. в главе 7). Затем Smss создает экземпляр Winlogon и Csrss для данного сеанса.

Winlogon, LSASS и Userinit

Процесс входа в Windows (\Windows\System32\Winlogon.exe) обрабатывает интерактивный вход пользователя в систему и выход из нее. При нажатии комбинации клавиш SAS (secure attention sequence) Winlogon получает уведомление о запросе пользователя на вход в систему. По умолчанию SAS в Windows представляет собой комбинацию клавиш Ctrl+Alt+Del. Назначение SAS — защита пользователя от программ перехвата паролей, имитирующих процесс входа в систему, так как эту комбинацию клавиш нельзя перехватить в приложении пользовательского режима.

Идентификация и аутентификация при входе в систему реализованы в заменяемой DLL под названием GINA (Graphical Identification and Authentication). Стандартная GINA Windows, Msgina.dll, реализует интерфейс для входа в систему по умолчанию. Однако разработчики могут включать свои GINA DLL, реализующие другие механизмы аутентификации и идентификации вместо стандартного метода Windows на основе проверки имени и пароля пользователя — например, на основе распознавания образцов голоса. Кроме того, Winlogon может загружать дополнительные DLL компонентов сетевого доступа для дальнейшей аутентификации. Эта функция позволяет нескольким компонентам доступа к сетям единовременно собирать все необходимые регистрационные данные в процессе обычного входа в систему.

После ввода имя и пароль пользователя посылаются для проверки серверному процессу локальной аутентификации (local security authentication server process, LSASS) (\Windows\System32\Lsass.exe, описываемому в главе 8). LSASS вызывает соответствующую функциональность (реализованную в виде DLL) для проверки соответствия введенного пароля с тем, что хранится в активном каталоге или SAM (части реестра, содержащей определения пользователей и групп).

После успешной аутентификации LSASS вызывает какую-либо функцию в мониторе состояния защиты (например, *NtCreateToken*), чтобы сгенерировать объект «маркер доступа» (access token object), содержащий профиль безопасности пользователя. Впоследствии Winlogon использует его для создания начального процесса оболочки. Информация о начальном процессе (или процессах) хранится в параметре Userinit в разделе реестра HKLM\ SOFTWARE\ Microsoft\Windows NT\CurrentVersion\Winlogon. (По умолчанию начальным процессом считается Userinit.exe, но в списке может быть более одного образа.)

Userinit выполняет некоторые действия по инициализации пользовательской среды (например, запускает сценарии регистрации и активизирует групповые политики), а затем ищет в реестре параметр Shell (в указанном выше разделе Winlogon) и создает процесс для запуска определенной системной оболочки (по умолчанию — Explorer.exe). После этого процесс Userinit завершается. Вот почему для Explorer.exe не показывается родительский процесс — он уже завершился. Иначе говоря, Explorer является «внучатым»

процессом Winlogon. (Имена процессов, чьи родительские процессы уже завершены, в списке Tlist выравниваются по левому краю.)

Winlogon активен не только при входе и выходе пользователя, но и при перехвате ввода SAS с клавиатуры. Например, когда вы нажимаете Ctrl+Alt+Del после входа в систему, Winlogon открывает диалоговое окно Windows Security (Безопасность Windows), предлагающее на выбор выход из системы, запуск Task Manager, блокировку рабочей станции, завершение работы системы и т. д.

Полное описание этапов процесса входа см. в разделе «Smss, Csrss и Winlogon» главы 5. Подробнее об аутентификации см. главу 8. Подробнее о вызываемых функциях интерфейса LSASS (чьи имена начинаются с *Lsa*) см. документацию Platform SDK.

Диспетчер управления сервисами (SCM)

Вспомните, что термин «сервисы» в Windows обозначает как серверные процессы, так и драйверы устройств. В этом разделе обсуждаются сервисы, являющиеся процессами пользовательского режима. Они похожи на демоны UNIX или обособленные процессы VMS в том смысле, что могут быть сконфигурированы на автоматический запуск при загрузке системы, не требуя интерактивного входа. Их также можно запустить вручную, например, с помощью оснастки Services (Службы) или вызовом Windows-функции *StartService*. Как правило, сервисы не взаимодействуют с вошедшим в систему пользователем, хотя при особых условиях это возможно (см. главу 4).

Этими сервисами управляет специальный системный процесс, диспетчер управления сервисами (service control manager) (\Windows\System32\Services.exe), отвечающий за запуск, остановку процессов сервисов и взаимодействие с ними. Сервисы представляют собой просто Windows-образы исполняемых программ, вызывающие особые Windows-функции для взаимодействия с диспетчером управления сервисами и с его помощью выполняющие такие операции, как регистрация успешного запуска сервиса, ответы на запросы о состоянии, приостановку или завершение работы сервиса. Сервисы определяются в разделе реестра HKLM\SYSTEM\CurrentControlSet\Services. Сведения о подразделах и параметрах, относящихся к сервисам, см. в справочном файле Regentry.chm в ресурсах Windows.

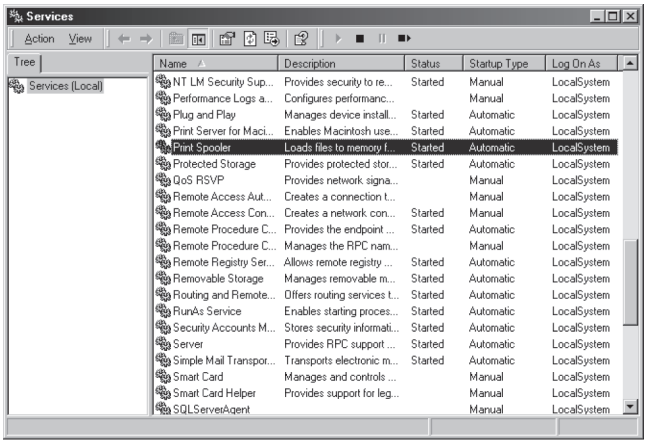
Учтите, что у сервисов есть три имени: имя процесса, выполняемого в системе, внутреннее имя в реестре и так называемое отображаемое имя (display name), которое можно увидеть в оснастке Services (Службы). (Не у каждого сервиса есть отображаемое имя, и в случае его отсутствия используется внутреннее имя.) Сервисы Windows также содержат поле описания, где находится более подробная информация о том, что делает конкретный сервис.

Чтобы выяснить, какие именно сервисы содержатся в том или ином процессе, введите команду **tlist /s**. Но заметьте, что иногда один процесс совместно используется несколькими сервисами. Код типа в реестре позволяет узнать, какие сервисы имеют собственные процессы и какие из них делят процессы с другими сервисами данного образа файла.

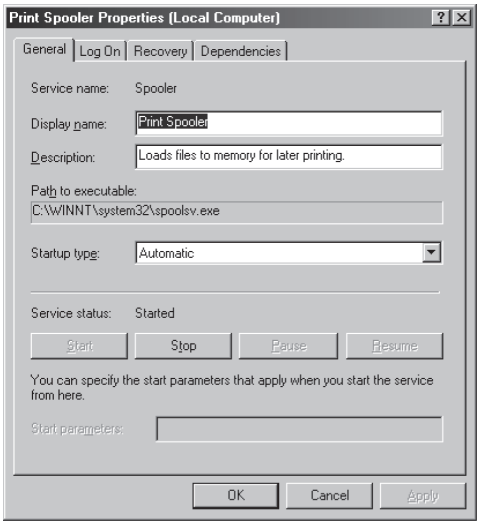
В виде сервисов реализуются некоторые компоненты Windows, например диспетчер очереди печати (спулер), журнал системных событий, планировщик задач, а также ряд сетевых компонентов.

ЭКСПЕРИМЕНТ: вывод списка установленных сервисов

Чтобы вывести список установленных сервисов (служб), дважды щелкните значок Administrative Tools (Администрирование) в окне Control Panel (Панель управления) и выберите Services (Службы). Вы должны увидеть что-нибудь в таком роде:



Для просмотра детальных сведений о сервисе щелкните правой кнопкой мыши имя сервиса и выберите команду Properties (Свойства). Ниже показан пример окна свойств для службы Print Spooler (Диспетчер очереди печати).

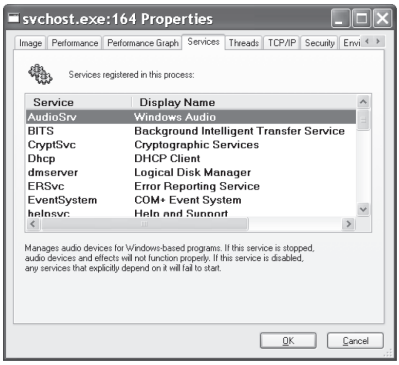


см. след. стр.

Обратите внимание, что поле Path To Executable (Исполняемый файл) указывает на программу, включающую данный сервис. Помните, что некоторые сервисы разделяют процессы с другими сервисами, поэтому число сервисов и используемых ими процессов не всегда находится в соотношении «один к одному».

ЭКСПЕРИМЕНТ: просмотр сервисов внутри сервисных процессов

Process Explorer выделяет процессы, которые являются хостами одного и более сервисов. (Для настройки поведения Process Explorer выберите Configure Highlighting в меню Options.) Дважды щелкнув процесс — хост сервисов, вы откроете вкладку Services, где перечисляются сервисы внутри этого процесса. При этом по каждому сервису выводится имя раздела реестра, где определен данный сервис, отображаемое имя, видимое администратору, и текст описания для этого сервиса (если такой текст есть). Например, в Windows XP список сервисов в процессе Svchost.exe, выполняемом под учетной записью System, выглядит следующим образом.



Подробнее о сервисах рассказывается в главе 4.

Резюме

В этой главе мы познакомились с общими аспектами системной архитектуры Windows. Мы также рассмотрели ключевые компоненты Windows и принципы их взаимодействия. В следующей главе будет подробнее рассказано о базовых системных механизмах, на которые опираются эти компоненты, в том числе о синхронизации и диспетчере объектов.

ГЛАВА 3

Системные механизмы

В Microsoft Windows существует несколько базовых механизмов, которыми пользуются компоненты режима ядра: исполнительная система (executive), ядро и драйверы устройств. В этой главе описываются следующие системные механизмы (а также способы их использования):

- диспетчеризация ловушек (trap dispatching), в том числе прерываний, DPC (deferred procedure call), APC (asynchronous procedure call), исключений и системных сервисов;
- диспетчер объектов исполнительной системы;
- синхронизация, в том числе спин-блокировки, объекты диспетчера ядра (kernel dispatcher objects) и реализация механизмов ожидания;
- системные рабочие потоки;
- различные механизмы вроде поддержки глобальных флагов Windows;
- LPC (local procedure call);
- Kernel Event Tracing;
- Wow64.

Диспетчеризация ловушек

Прерывания и исключения — такие ситуации в операционной системе, в которых нормальный поток выполнения кода процессором прерывается. Эти ситуации обнаруживаются как программным, так и аппаратным обеспечением. Термин *ловушка* (trap) относится к механизму, благодаря которому при прерывании или исключении процессор перехватывает контроль над выполняемым потоком и передает управление определенной части операционной системы. В Windows процессор передает управление *обработчику ловушек* (trap handler) — функции, специфичной для конкретного прерывания или исключения. Рис. 3-1 иллюстрирует некоторые ситуации, в которых активизируются обработчики ловушек.

Рис. 3-1. *Диспетчеризация ловушек*

Ядро различает прерывания и исключения: *прерывание* (interrupt) является асинхронным событием (т. е. оно может произойти в любой момент независимо от текущих команд, выполняемых процессором). Прерывания в основном генерируются устройствами ввода-вывода и таймерами. Их можно включать и отключать. *Исключение* (exception), напротив, представляет собой синхронное событие, являющееся результатом выполнения конкретной команды. Повторный запуск программы в аналогичных условиях с теми же данными позволит воспроизвести исключение. Примерами исключений могут служить нарушения доступа (ошибки защиты памяти), выполнение некоторых команд отладчика, а также попытки деления на нуль. Ядро также считает исключениями вызовы системных сервисов (хотя с точки зрения технической реализации это системные ловушки).

Прерывания и исключения можно генерировать как программно, так и аппаратно. Например, исключение «bus error» (ошибка шины) возникает из-за аппаратной ошибки, а причиной исключения «divide-by-zero» (деление на нуль) является ошибка в программе. Аналогичным образом прерывания могут генерироваться устройствами ввода-вывода или самим ядром (такие программные прерывания, как, например, APC или DPC).

При аппаратном прерывании или исключении процессор записывает статусную информацию в стек ядра для прерванного потока, чтобы впоследствии можно было вернуться к исходной точке в потоке управления и продолжить выполнение команд так, будто ничего не произошло. Если поток выполнялся в пользовательском режиме, Windows переключается на стек режима ядра для потока. Затем создает в стеке ядра прерванного потока

фрейм ловушки (trap frame), в котором сохраняет информацию о состоянии потока. Фрейм ловушки является подмножеством полного контекста потока (см. главу 6), и вы можете просмотреть его определение, введя в отладчике ядра команду **dt nt!_ktrap_frame**. Программное прерывание ядро обслуживает либо при обработке аппаратного прерывания, либо синхронно — при вызове потоком функции ядра, относящейся к данному программному прерыванию.

В большинстве случаев ядро устанавливает функции, выполняющие общую обработку ловушек до и после передачи управления другим функциям, которые ставят ловушки. Например, когда устройство генерирует прерывание, обработчик ловушек аппаратных прерываний (принадлежащий ядру) передает управление *процедуре обслуживания прерывания* (interrupt service routine, ISR), предоставленной драйвером соответствующего устройства. Если прерывание возникло в результате вызова системного сервиса, обработчик ловушек общесистемных сервисов передает управление функции указанного системного сервиса в исполнительной системе. Ядро также устанавливает обработчики для ловушек, которые оно не ожидает или не обрабатывает. Эти обработчики, как правило, выполняют системную функцию *KeBugCheckEx*. Она останавливает компьютер, если ядро обнаруживает в работе системы отклонения, способные привести к повреждению данных (подробнее об этом см. главу 14). Диспетчеризация прерываний, исключений и системных сервисов детальнее описывается в следующих разделах.

Диспетчеризация прерываний

Аппаратные прерывания обычно генерируются устройствами ввода-вывода, которые таким образом уведомляют процессор о необходимости уделить им внимание. Устройства, управляемые на основе прерываний, позволяют операционной системе максимально полно использовать процессор, совмещая основную обработку с обслуживанием ввода-вывода. Выдав запрос на ввод-вывод, поток может заняться другой работой, пока устройство выполняет запрошенную операцию. Закончив, устройство генерирует прерывание, и процессор переключается на обслуживание этого устройства. Прерываниями управляются, как правило, координатные устройства, принтеры, клавиатуры, дисковые устройства и сетевые платы.

Системное программное обеспечение также может генерировать прерывания. Ядро способно отключать прерывания, чтобы не прерывать работу процессора, однако это делается нечасто — только в критические моменты, например при обработке прерываний или диспетчеризации исключения.

Для обработки аппаратных прерываний ядро устанавливает обработчики ловушек прерываний, которые передают управление внешней процедуре (ISR), обрабатывающей прерывание, или внутренней процедуре ядра, реагирующей на прерывание. Драйверы устройств предоставляют ISR для обслуживания прерываний от своих устройств, а ядро — внутренние процедуры для обработки других типов прерываний.

Далее мы рассмотрим, как процессор уведомляется об аппаратных прерываниях, какие типы прерываний поддерживаются ядром и как драйверы устройств взаимодействуют с ядром (в процессе обработки прерываний). Кроме того, мы поговорим о распознавании ядром программных прерываний и об объектах, используемых для реализации таких прерываний.

Обработка аппаратных прерываний

На аппаратных платформах, поддерживаемых Windows, прерывания, связанные с внешним вводом-выводом, поступают по одной из линий контроллера прерываний. Контроллер в свою очередь связан с процессором единственной линией, по которой и уведомляет о прерывании. Как только процессор прерывается, он требует от контроллера запрос прерывания (interrupt request, IRQ). Контроллер транслирует IRQ в номер прерывания, используемый как индекс в структуре, называемой *таблицей диспетчеризации прерываний* (interrupt dispatch table, IDT), и передает управление соответствующей процедуре. При загрузке Windows заносит в IDT указатели на процедуры ядра, обрабатывающие каждое прерывание и исключение.

ЭКСПЕРИМЕНТ: просмотр IDT

Просмотреть содержимое IDT, включая сведения об обработчиках ловушек, которые Windows назначила прерываниям, можно с помощью команды *!idt* отладчика ядра. Команда *!idt* без флагов показывает векторы, которые сопоставлены с адресами в модулях, отличных от Ntoskrnl.exe.

Ниже показано, что выводит команда *!idt*.

```
kd> !idt

Dumping IDT:

30:     806b14c0 hal!HalpClockInterrupt
31:     8a39dc3c i8042prt!I8042KeyboardInterruptService
                 (KINTERRUPT 8a39dc00)
34:     8a436dd4 serial!SerialCIsrSw (KINTERRUPT 8a436d98)
35:     8a44ed74 NDIS!ndisMIsr (KINTERRUPT 8a44ed38)
                 portcls!CInterruptSync::Release+0x10
                 (KINTERRUPT 899c44a0)
38:     806abe80 hal!HalpProfileInterrupt
39:     8a4a8abc ACPI!ACPIInterruptServiceRoutine (KINTERRUPT
                 8a4a8a80)
3b:     8a48d8c4 pcmcia!PcmciaInterrupt (KINTERRUPT 8a48d888)
                 ohci1394!OhciIsr (KINTERRUPT 8a41da18)
                 VIDEOPRT!pVideoPortInterrupt (KINTERRUPT
                 8a1bc2c0)
                 USBPORT!USBPORT_InterruptService (KINTERRUPT
                 8a2302b8)
                 USBPORT!USBPORT_InterruptService (KINTERRUPT
```

```
                8a0b8008)
                USBPORT!USBPORT_InterruptService (KINTERRUPT
                8a170008)
                USBPORT!USBPORT_InterruptService (KINTERRUPT
                8a258380)
                NDIS!ndisMIsr (KINTERRUPT 8a0e0430)
3c:     8a39d3ec i8042prt!I8042MouseInterruptService (KINTERRUPT
                8a39d3b0)
3e:     8a47264c atapi!IdePortInterrupt (KINTERRUPT 8a472610)
3f:     8a489b3c atapi!IdePortInterrupt (KINTERRUPT 8a489b00)
```

В системе, задействованной в этом эксперименте, номер прерывания 0x3C — с ISR драйвера клавиатуры (I8042prt.sys), а прерывание 0x3B совместно используется несколькими устройствами, в том числе видеоадаптером, шиной PCMCIA, портами USB и IEEE 1394, а также сетевым адаптером.

Windows увязывает аппаратные IRQ с номерами прерываний в IDT. Эта таблица используется системой и при конфигурировании обработчиков ловушек для исключений. Так, номер исключения для ошибки страницы на x86 и x64 (это исключение возникает, когда поток пытается получить доступ к отсутствующей или не определенной в виртуальной памяти странице) равен 0xe. Следовательно, запись 0xe в IDT указывает на системный обработчик ошибок страниц. Хотя архитектуры, поддерживаемые Windows, допускают до 256 элементов в IDT, число IRQ на конкретной машине определяется архитектурой используемого в ней контроллера прерываний.

У каждого процессора имеется своя IDT, так что разные процессоры могут при необходимости выполнять разные ISR. Например, в многопроцессорной системе каждый процессор получает прерывания системного таймера, но обновление значения системного таймера в результате обработки этого прерывания осуществляется только одним процессором. Однако все процессоры используют это прерывание для измерения кванта времени, выделенного потоку, и для инициации новой процедуры планирования по истечении этого кванта. Аналогичным образом в некоторых конфигурациях может понадобиться, чтобы определенные аппаратные прерывания обрабатывал конкретный процессор.

Контроллеры прерываний на платформе x86

В большинстве систем x86 применяется либо программируемый контроллер прерываний (Programmable Interrupt Controller, PIC) i8259A, либо его разновидность, усовершенствованный программируемый контроллер прерываний (Advanced Programmable Interrupt Controller, APIC) i82489. Новые компьютеры, как правило, оснащаются APIC. Стандарт PIC был разработан для оригинальных IBM PC. PIC работает только в однопроцессорных системах и имеет 15 линий прерываний. APIC и SAPIC (о нем чуть позже) способен работать в многопроцессорных системах и предлагает 256 линий пре-

рываний. Intel совместно с другими компаниями создали спецификацию Multiprocessor (MP) Specification, стандарт для многопроцессорных систем x86, основанный на использовании APIC. Для совместимости с однопроцессорными операционными системами и загрузочным кодом, запускающим многопроцессорную систему в однопроцессорном режиме, APIC поддерживает PIC-совместимый режим с 15 линиями прерываний и передачей прерываний лишь главному процессору. Архитектура APIC показана на рис. 3-2. На самом деле APIC состоит из нескольких компонентов: APIC ввода-вывода, принимающего прерывания от устройств, локальных APIC, принимающих прерывания от APIC ввода-вывода по выделенной шине и прерывающих работу того процессора, с которым они связаны, а также i8259A-совместимого контроллера прерываний, транслирующего входные сигналы APIC в соответствующие PIC-эквиваленты. APIC ввода-вывода отвечает за реализацию алгоритмов перенаправления прерываний, и операционная система выбирает нужный ей алгоритм (в Windows выбор возлагается на HAL). Эти алгоритмы равномерно распределяют между процессорами нагрузку, связанную с обработкой прерываний от устройств, и в максимальной мере используют все преимущества локальности, направляя такие прерывания процессору, который только что обрабатывал прерывания аналогичного типа.

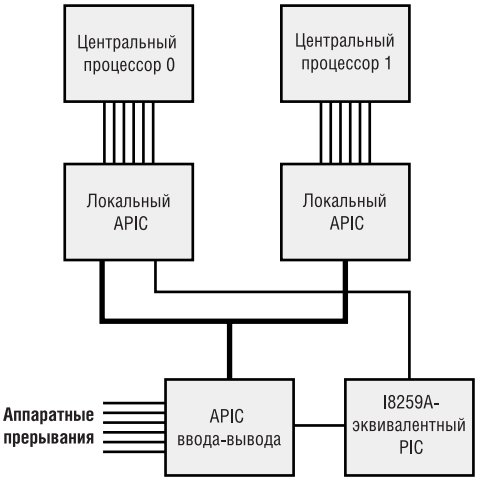


Рис. 3-2. *Архитектура x86 APIC*

Контроллеры прерываний на платформе x64

Поскольку архитектура x64 совместима с операционными системами для x86, системы на базе x64 должны предоставлять те же контроллеры прерываний, что и на базе x86. Однако x64-версии Windows не будут работать на системах без APIC (т. е. они не поддерживают PIC).

Контроллеры прерываний на платформе IA64

В архитектуре IA64 используется контроллер прерываний Streamlined Advanced Programmable Interrupt Controller (SAPIC) — результат эволюционного развития APIC. Главное различие между архитектурами APIC и SAPIC в том, что APIC ввода-вывода в APIC-системе направляет прерывания локальным APIC по выделенной шине APIC, тогда как в системе SAPIC прерывания передаются по шине ввода-вывода и системы (I/O and system bus) для большего быстродействия. Еще одно различие — перенаправление прерываний и балансировка нагрузки в APIC-системе обрабатывается самой шиной APIC, а в SAPIC-системе, где нет выделенной шины APIC, требуется, чтобы соответствующая поддержка была запрограммирована в микрокоде (прошивке). Но, даже если эта поддержка имеется в микрокоде, Windows не использует ее — вместо этого она статически назначает прерывания процессорам по принципу карусели.

ЭКСПЕРИМЕНТ: просмотр конфигурации PIC и APIC

Конфигурацию PIC в однопроцессорной системе и APIC в многопроцессорной системе можно просмотреть с помощью команд *!pic* или *!apic* отладчика ядра. (Для этого эксперимента LiveKd не годится, так как она не может напрямую обращаться к оборудованию.) Ниже показан образец вывода команды *!pic* в однопроцессорной системе (учтите, что команда *!pic* не работает в системе, использующей APIC HAL).

```
lkd> !pic
-- IRQ Number -- 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
Physically in service:  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
Physically masked:      .  .  .  Y  .  .  Y  Y  .  .  Y  .  .  Y  .  .
Physically requested:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
Level triggered:        .  .  .  .  .  Y  .  .  .  Y  .  Y  .  .  .  .
```

На следующем листинге приводится выходная информация команды *!apic* в системе, использующей MPS HAL. Префикс «0:» в командной строке отладчика говорит о том, что текущие команды выполняются на процессоре 0, поэтому данный листинг относится к APIC ввода-вывода процессора 0.

```
lkd> !apic
Apic @ fffe0000  ID:0 (40010) LogDesc:01000000 DestFmt:ffffffff  TPR 20
TimeCnt: 0bebc200clk  SpurVec:3f  FaultVec:e3  error:0
Ipi Cmd: 0004001f  Vec:1F  FixedDel    Dest=Self      edg  high
Timer..: 000300fd  Vec:FD  FixedDel    Dest=Self      edg  high
masked
Linti0.: 0001003f  Vec:3F  FixedDel    Dest=Self      edg  high
masked
Linti1.: 000184ff  Vec:FF  NMI         Dest=Self      lvl  high
masked
TMR: 61, 82, 91-92, B1
IRR:
ISR:
```

см. след. стр.

Теперь взгляните на образец вывода команды *!ioapic*, показывающей конфигурацию APIC ввода-вывода:

```
0: kd> !ioapic
IoApic @ ffd02000  ID:8 (11)  Arb:0
Inti00.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti01.: 00000962  Vec:62  LowestDl  Lg:03000000  edg
Inti02.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti03.: 00000971  Vec:71  LowestDl  Lg:03000000  edg
Inti04.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti05.: 00000961  Vec:61  LowestDl  Lg:03000000  edg
Inti06.: 00010982  Vec:82  LowestDl  Lg:02000000  edg  masked
Inti07.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti08.: 000008d1  Vec:D1  FixedDel  Lg:01000000  edg
Inti09.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti0A.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti0B.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti0C.: 00000972  Vec:72  LowestDl  Lg:03000000  edg
Inti0D.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti0E.: 00000992  Vec:92  LowestDl  Lg:03000000  edg
Inti0F.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti10.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti11.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti12.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti13.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti14.: 0000a9a3  Vec:A3  LowestDl  Lg:03000000  lvl
Inti15.: 0000a993  Vec:93  LowestDl  Lg:03000000  lvl
Inti16.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti17.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
```

Уровни запросов программных прерываний

Хотя контроллеры прерываний различают уровни приоритетов прерываний, Windows использует свою схему приоритетов прерываний, известную под названием *уровни запросов прерываний* (interrupt request levels, IRQL). Внутри ядра IRQL представляются в виде номеров 0–31 в системах x86 и 0–15 в системах x64 и IA64, причем больший номер соответствует прерыванию с более высоким приоритетом. Ядро определяет стандартный набор IRQL для программных прерываний, а HAL увязывает IRQL с номерами аппаратных прерываний. IRQL, определенные для архитектуры x86, показаны на рис. 3-3, а аналогичные сведения для архитектур x64 и IA64 — на рис. 3-4.

ПРИМЕЧАНИЕ Уровень SYNCH_LEVEL, используемый многопроцессорными версиями ядра для защиты доступа к индивидуальным для каждого процессора блокам PRCB (processor control blocks), не показан на этих схемах, так как его значение варьируется в разных версиях Windows. Описание SYNCH_LEVEL и его возможных значений см. в главе 6.

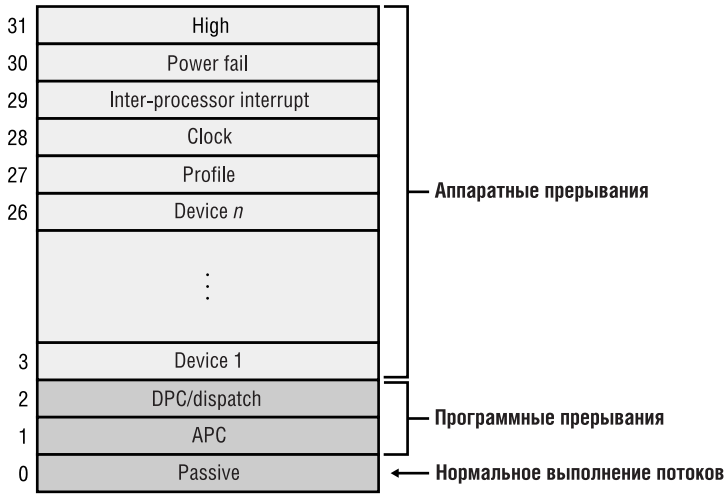


Рис. 3-3. *Уровни запросов прерываний (IRQL) в x86-системах*

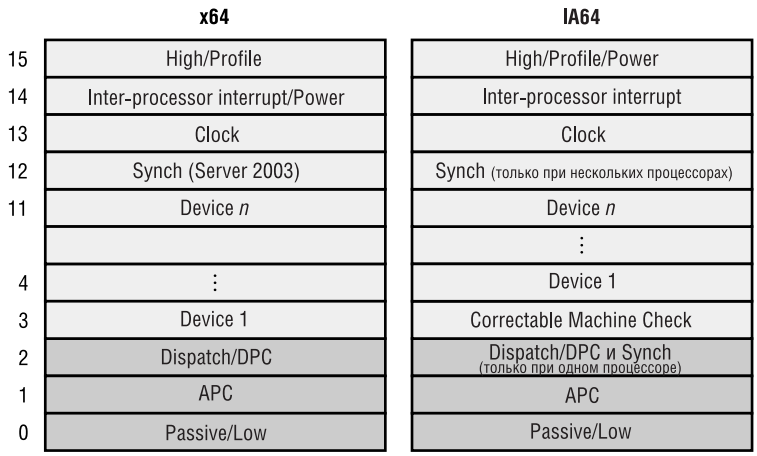


Рис. 3-4. *Уровни запросов прерываний (IRQL) в системах x64 и IA64*

Прерывания обслуживаются в порядке их приоритета, и прерывания с более высоким приоритетом вытесняют обработку прерываний с меньшим приоритетом. При возникновении прерывания с высоким приоритетом процессор сохраняет информацию о состоянии прерванного потока и активизирует сопоставленный с данным прерыванием диспетчер ловушки. Последний повышает IRQL и вызывает процедуру обслуживания прерывания (ISR). После выполнения ISR диспетчер прерывания понижает IRQL процессора до исходного уровня и загружает сохраненные ранее данные о состоянии машины. Прерванный поток возобновляется с той точки, где он был прерван. Когда ядро понижает IRQL, могут «материализоваться» ранее замаскированные прерывания с более низким приоритетом. Тогда вышеописанный процесс повторяется ядром для обработки и этих прерываний.

Уровни приоритетов IRQL имеют совершенно иной смысл, чем приоритеты в схеме планирования потоков (см. главу 6). Приоритет в этой схеме является атрибутом потока, тогда как IRQL — атрибутом источника прерывания, например клавиатуры или мыши. Кроме того, IRQL каждого процессора меняется во время выполнения команд операционной системы.

Значение IRQL определяет, какие прерывания может получать данный процессор. IRQL также используется для синхронизации доступа к структурам данных режима ядра (о синхронизации мы поговорим позже). При выполнении поток режима ядра повышает или понижает IRQL процессора либо напрямую (вызовом соответственно *KeRaiseIrql* или *KeLowerIrql*), либо — что бывает гораздо чаще — опосредованно (через функции, которые обращаются к синхронизирующим объектам ядра). Как показано на рис. 3-5, прерывания от источника с IRQL, превышающим текущий уровень, прерывают работу процессора, а прерывания от источников, IRQL которых меньше или равен текущему уровню, *маскируются* до тех пор, пока выполняемый поток не понизит IRQL.

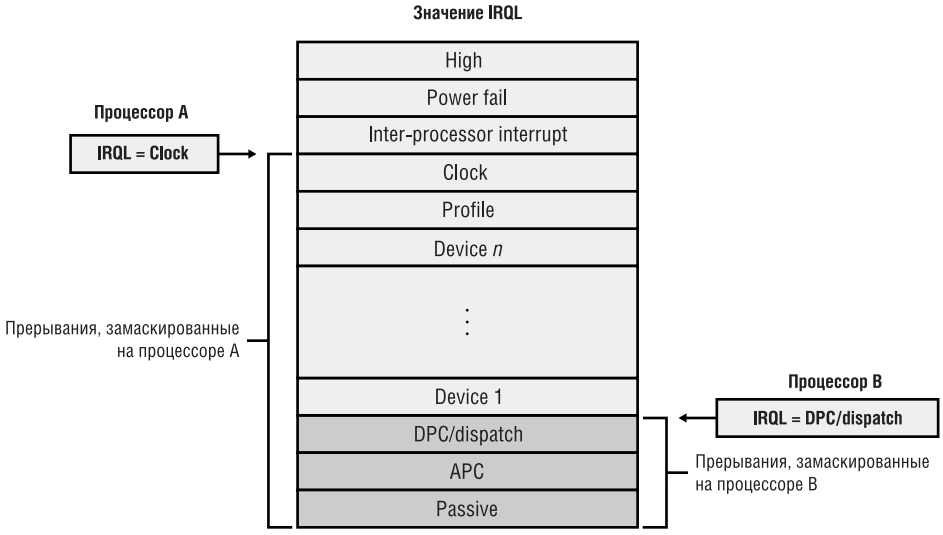


Рис. 3-5. *Маскирование прерываний*

Поскольку доступ к PIC — операция довольно медленная, в HAL, использующих PIC, реализован механизм оптимизации «отложенный IRQL» (lazy IRQL), который избегает обращений к PIC. Когда IRQL повышается, HAL — вместо того чтобы изменять маску прерывания — просто отмечает новый IRQL. Если вслед за этим возникает прерывание с более низким приоритетом, HAL устанавливает маску прерывания в соответствии с первым и откладывает обработку прерывания с более низким приоритетом до понижения IRQL. Таким образом, если при повышенном IRQL не возникнет прерываний с более низким приоритетом, HAL не потребуется обращаться к PIC.

Поток режима ядра повышает и понижает IRQL процессора, на котором он выполняется, в зависимости от того, что именно делает этот поток. Например, обработчик ловушки (или сам процессор) при прерывании повышает IRQL процессора до IRQL источника прерывания. В результате все прерывания с более низким или равным IRQL маскируются (только на этом процессоре), что не дает прерыванию с таким же или более низким IRQL помешать процессору обработать текущее прерывание. Замаскированные прерывания либо обрабатываются другим процессором, либо откладываются до понижения IRQL. Поэтому все системные компоненты, в том числе ядро и драйверы устройств, пытаются удерживать IRQL на уровне passive («пассивный»), иногда называемом низким уровнем. Если бы IRQL долго оставался неоправданно высоким, драйверы устройств не смогли бы оперативно реагировать на аппаратные прерывания.

ПРИМЕЧАНИЕ Прерывания APC_LEVEL являются исключением из правила, которое гласит, что повышение IRQL блокирует прерывания такого же уровня и ниже. Если поток повышает IRQL до уровня APC_LEVEL, а затем отключается от процессора из-за появления прерывания DISPATCH_LEVEL, то система может доставить ему прерывание APC_LEVEL, как только он вновь получит процессорное время. Таким образом, APC_LEVEL можно считать IRQL, локальным для потока.

ЭКСПЕРИМЕНТ: определяем IRQL

Если вы работаете с отладчиком ядра в Windows Server 2003, то можете определить IRQL процессора командой *!irql*:

```
kd> !irql
Debugger saved IRQL for processor 0x0 - 0 (LOW_LEVEL)
```

Заметьте, что в структуре данных, называемой PCR (processor control region), и ее расширении — PRCB (processor control block) имеется поле с именем Irql. Эти структуры содержат информацию о состоянии каждого процессора в системе, в том числе текущий IRQL, указатель на аппаратную IDT, сведения о текущем потоке и потоке, который будет выполняться следующим. Ядро и HAL используют эту информацию для выполнения операций, специфичных для данной машины и ее архитектуры. Отдельные части структур PCR и PRCB открыто определены в заголовочном файле Ntddk.h (в Windows DDK). Загляните в него, чтобы получить представление об этих структурах.

Для просмотра содержимого PCR воспользуемся командой *!pcr* отладчика ядра.

```
kd> !pcr
PCR Processor 0 @ffdff000
        NtTib.ExceptionList: f8effc68
            NtTib.StackBase: f8effdf0
           NtTib.StackLimit: f8efd000
```

см. след. стр.

```
NtTib.SubSystemTib: 00000000
     NtTib.Version: 00000000
 NtTib.UserPointer: 00000000
     NtTib.SelfTib: 7ffde000

           SelfPcr: ffdff000
              Prcb: ffdff120
              Irql: 00000000
               IRR: 00000000
               IDR: ffff28e8
     InterruptMode: 00000000
               IDT: 80036400
               GDT: 80036000
               TSS: 802b5000

     CurrentThread: 81638020
        NextThread: 00000000
        IdleThread: 8046bdf0
```

К сожалению, Windows не поддерживает поле Irql на платформах, не использующих отложенные IRQL, поэтому в большинстве систем это поле всегда содержит 0.

Так как изменения IRQL процессора существенно влияют на функционирование системы, они возможны только в режиме ядра. Потоки пользовательского режима не могут изменять IRQL процессора. Это значит, что при выполнении потоков пользовательского режима значение IRQL процессора всегда равно passive. Только при выполнении кода режима ядра IRQL может быть выше этого уровня.

Каждый уровень прерывания имеет определенное назначение. Так, ядро генерирует *межпроцессорное прерывание* (interprocessor interrupt, IPI), чтобы потребовать выполнения какой-либо операции от другого процессора, например, при диспетчеризации некоего потока или обновлении кэша ассоциативного буфера трансляции [translation look-aside buffer (TLB) cache]. Системный таймер через регулярные промежутки генерирует прерывания, на которые ядро реагирует обновлением системного времени, и это используется для измерения продолжительности выполнения потока. Если аппаратная платформа поддерживает два таймера, то для измерения производительности ядро добавляет еще один уровень прерываний от таймера. HAL поддерживает несколько уровней запросов прерываний для устройств, управляемых прерываниями; конкретное число таких уровней зависит от процессора и конфигурации системы. Ядро использует программные прерывания для инициации планирования потоков и асинхронного вмешательства в выполнение потока.

Увязка прерываний с IRQL Уровни IRQL и запросы прерываний (IRQ) — вещи разные. Концепция IRQL в архитектурах, на которых работает Windows, не реализована аппаратно. Тогда возникает вопрос: как Windows оп-

ределяет, какой IRQL следует присвоить прерыванию? Ответ нужно искать в HAL. В Windows за определение устройств на конкретной шине (PCI, USB и т. д.) и назначение им прерываний отвечают драйверы устройств особого типа — драйверы шин. *Драйвер шины* сообщает эту информацию диспетчеру Plug and Play, и тот, учитывая приемлемые для других устройств прерывания, принимает решение о конкретных прерываниях, выделяемых каждому устройству. Далее он вызывает HAL-функцию *HalpGetSystemInterruptVector*, которая увязывает прерывания со значениями IRQL.

Этот алгоритм неодинаков в различных версиях HAL. В однопроцессорных x86-системах HAL выполняет прямую трансляцию: IRQL данного вектора прерывания вычисляется путем вычитания значения вектора из 27. Таким образом, если устройство использует 5-й вектор прерывания, его ISR выполняется при IRQL, равном 22. В многопроцессорной x86-системе преобразования более сложны. APIC поддерживает более 200 векторов прерываний, поэтому при трансляции «один в один» имеющихся IRQL окажется недостаточно. Многопроцессорная версия HAL присваивает IRQL векторам прерываний, циклически перебирая значения из диапазона IRQL устройств (device IRQL, DIRQL). В итоге на многопроцессорной x86-системе не так-то просто предсказать или выяснить IRQL, назначаемый IRQ. Наконец, в x64- и IA64-системах HAL вычисляет IRQL для IRQ путем деления вектора прерывания, назначенного данному IRQ, на 16.

Предопределенные IRQL Давайте повнимательнее приглядимся к предопределенным IRQL, начиная с самого верхнего уровня схемы, представленной на рис. 3-5.

- Уровень «high» (высокий) используется ядром, только если оно останавливает систему в функции *KeBugCheckEx* и маскирует все прерывания.
- Уровень «power fail» (отказ электропитания) был заложен еще в самый первый проект Microsoft Windows NT. Он определяет поведение системы при отказе электропитания, но никогда не применялся.
- Уровень «interprocessor interrupt» (межпроцессорное прерывание) используется для того, чтобы запрашивать от другого процессора выполнение какой-либо операции, например, при постановке в очередь прерывания DISPATCH_LEVEL для планирования конкретного потока к выполнению, при обновлении кэша TLB, завершении работы или крахе системы.
- Уровень «clock» (часы) используется для системных часов, с помощью которых ядро отслеживает время суток, измеряет и распределяет процессорное время между потоками.
- Уровень «profile» (профиль) используется системным таймером реального времени, если активизирован механизм профилирования ядра (kernel profiling), т. е. измерения его производительности. Когда он активен, обработчик ловушки профилирования регистрирует адрес команды, выполнявшейся на момент прерывания. Со временем создается таблица адресов, которую можно извлечь и проанализировать с помощью соответствующих утилит. Вы можете скачать утилиту Kernrate (*http://www.microsoft.*

com/whdc/system/sysperf/krview.mspx), позволяющую просматривать статистику, полученную при использовании механизма профилирования ядра. Подробнее об этой утилите см. описание эксперимента с Kernrate.

- Уровень «device» (устройство) применяется для задания приоритетов прерываний от устройств (о принципах увязки аппаратных прерываний с IRQL см. предыдущий раздел).
- Прерывания уровней «DPC/dispatch» и «APC» являются программными; они генерируются ядром и драйверами устройств (о DPC и APC будет рассказано позже).
- Самый низкий уровень IRQL, «passive» (пассивный), на самом деле вообще не является уровнем прерывания. При этом значении IRQL потоки выполняются обычным образом и могут возникать любые прерывания.

ЭКСПЕРИМЕНТ: применение утилиты Kernrate

Утилита профилирования ядра (Kernrate) позволяет включать таймер профилирования системы, собирать образцы кода, выполняемого при срабатывании таймера, и выводить сводную информацию, отражающую распределение процессорного времени по образам файлов и функциям. Ее можно использовать для отслеживания процессорного времени, потребляемого индивидуальными процессами, и/или времени, проведенного в режиме ядра независимо от процессов (например, для выполнения процедур обслуживания прерываний). Профилирование ядра полезно, когда вы хотите выявить точки, в которых на выполнение кода тратится больше всего процессорного времени.

В своей простейшей форме Kernrate сообщает, сколько процессорного времени было использовано каждым модулем ядра (Ntoskrnl, драйверами и т. д.). Попробуйте, к примеру, выполнить следующие операции.

1. Откройте окно командной строки.
2. Введите **cd c:\program files\krview\kernrates**.
3. Введите **dir**. (Вы увидите образы kernrate для каждой платформы.)
4. Запустите образ, который подходит для вашей платформы (без аргументов или ключей). Например, Kernrate_i386_XP.exe — это образ для Windows XP на платформе x86.
5. Пока Kernrate выполняется, поделайте что-нибудь в системе. Скажем, запустите Windows Media Player и проиграйте музыку, запустите игру, интенсивно работающую с графикой, или перечислите содержимое каталога на удаленном сетевом ресурсе.
6. Нажмите Ctrl+C, чтобы остановить Kernrate. Это заставит Kernrate вывести статистику за прошедший период.

Ниже приведена часть вывода Kernrate, когда выполнялся Windows Media Player, воспроизводивший дорожку с компакт-диска.

```
C:\Program Files\KrView\Kernrates>Kernrate_i386_XP.exe
 /==============================\
<         KERNRATE LOG           >
 \==============================/
Date: 2004/05/13   Time:  9:48:28
Machine Name: BIGDAVID
Number of Processors: 1
PROCESSOR_ARCHITECTURE: x86
PROCESSOR_LEVEL: 6

Kernrate User-Specified Command Line:
Kernrate_i386_XP.exe

***> Press ctrl-c to finish collecting profile data
===> Finished Collecting Data, Starting to Process Results

------Overall Summary:-------
P0     K 0:00:03.234 (11.7%)  U 0:00:08.352 (30.2%)  I 0:00:16.093 (58.1%)
 DPC 0:00:01.772 ( 6.4%)  Interrupt 0:00:00.350 ( 1.3%)
       Interrupts= 52899, Interrupt Rate= 1911/sec.
Time   7315 hits, 19531 events per hit ----
 Module                                  Hits      msec   %Total    Events/Sec
gv3                                     4735      27679    64 %     3341135
smwdm                                    872      27679    11 %      615305
win32k                                   764      27679    10 %      539097
ntoskrnl                                 739      27679    10 %      521457
hal                                      124      27679     1 %       87497
```

Сводные данные показывают, что система провела 11,7% времени в режиме ядра, 30,2% в пользовательском режиме, 58,1% в простое, 6,4% на уровне DPC и 1,3% на уровне прерываний (interrupt level). Модуль, чаще всего требовавший к себе внимания, был GV3.SYS, драйвер процессора для Pentium M (семейства Geyserville). Он используется для сбора информации о производительности, поэтому и оказался на первом месте. Модуль, занявший второе место, — Smwdm.sys, драйвер звуковой платы на компьютере, где проводился тест. Это вполне объяснимо, учитывая, что основную нагрузку в системе создавал Windows Media Player, посылавший звуковой ввод-вывод этому драйверу.

Если у вас есть файлы символов, вы можете исследовать индивидуальные модули и посмотреть, сколько времени было затрачено каждой из их функций. Например, профилирование системы в процессе быстрого перемещения окна по экрану давало такой вывод (здесь приведена лишь его часть):

```
C:\Program Files\KrView\Kernrates>Kernrate_i386_XP.exe -z
  ntoskrnl -z win32k
```

см. след. стр.

```
 /============================\
<         KERNRATE LOG         >
 \============================/
Date: 2004/05/13   Time: 10:26:55

Time   4087 hits, 19531 events per hit ----
 Module                                Hits      msec   %Total    Events/Sec
win32k                                1649      10424    40 %     3089660
ati2dvag                              1269      10424    31 %     2377670
ntoskrnl                               794      10424    19 %     1487683
gv3                                    162      10424     3 %      303532

-- Zoomed module win32k.sys (Bucket size = 16 bytes, Rounding Down) ----
 Module                               Hits      msec   %Total    Events/Sec
EngPaint                               328      10424    19 %      614559
EngLpkInstalled                        302      10424    18 %      565844

-- Zoomed module ntoskrnl.exe (Bucket size = 16 bytes, Rounding Down) ---
Module                                Hits      msec   %Total    Events/Sec
KiDispatchInterrupt                    243      10424    26 %      455298
ZwYieldExecution                        50      10424     5 %       93682
InterlockedDecrement                    39      10424     4 %       73072
```

В данном случае самым «прожорливым» был модуль Win32k.sys, драйвер системы, отвечающей за работу с окнами. Второй в списке — видеодрайвер. И действительно, основная нагрузка в системе была связана с рисованием окна на экране. В детальном выводе для Win32k.sys видно, что наиболее активна была его функция EngPaint, основная GDI-функция для рисования на экране.

На код, выполняемый на уровне «DPC/dispatch» и выше, накладывается важное ограничение: он не может ждать освобождения объекта, если такое ожидание заставило бы планировщик подключить к процессору другой поток (а это недопустимая операция, так как планировщик синхронизирует свои структуры данных на уровне «DPC/dispatch» и, следовательно, не может быть активизирован для выполнения перераспределения процессорного времени). Другое ограничение заключается в том, что при уровне IRQL «DPC/dispatch» или выше доступна только неподкачиваемая память. На самом деле второе ограничение является следствием первого, так как обращение к отсутствующей в оперативной памяти странице вызывает ошибку страницы. Тогда диспетчер памяти должен был бы инициировать операцию дискового ввода-вывода, после чего ждать, когда драйвер файловой системы загрузит эту страницу с диска. Это в свою очередь вынудило бы планировщик переключить контекст (возможно, на поток простоя, если нет ни одного пользовательского потока, ждущего выполнения). В результате было бы нарушено правило, запрещающее вызов планировщика в таких ситуациях (поскольку при чтении с диска IRQL все еще остается на уровне «DPC/dispatch»

или выше). При нарушении любого из этих двух ограничений происходит крах системы с кодом завершения IRQL_NOT_LESS_OR_EQUAL (подробнее о кодах завершения при крахе системы см. главу 4). Кстати, нарушение этих ограничений является довольно распространенной ошибкой в драйверах устройств. Локализовать причину ошибок такого типа помогает утилита Driver Verifier, о которой будет подробно рассказано в разделе «Утилита Driver Verifier» главы 7.

Объекты «прерывание» (interrupt objects) Ядро предоставляет переносимый (портируемый) механизм — объект прерывания, позволяющий драйверам устройств регистрировать ISR для своих устройств. Этот объект содержит всю информацию, необходимую ядру для назначения конкретного уровня прерывания для ISR устройства, включая адрес ISR, IRQL устройства и запись в IDT ядра, с которой должна быть сопоставлена ISR. При инициализации в объект прерывания из шаблона обработки прерываний, *KiInterruptTemplate*, копируется несколько ассемблерных инструкций — код диспетчеризации. Этот код выполняется при возникновении прерывания.

Код, хранящийся в объекте прерывания, вызывает реальный диспетчер прерываний, которым обычно является процедура ядра *KiInterruptDispatch* или *KiChainedDispatch*, и передает ему указатель на объект прерывания. *KiInterruptDispatch* обслуживает векторы прерываний, для которых зарегистрирован только один объект прерывания, а *KiChainedDispatch* — векторы, разделяемые несколькими объектами прерываний. В объекте прерывания содержится информация, необходимая процедуре *KiChainedDispatch* для поиска и корректного вызова ISR драйвера. Объект прерывания также хранит значение IRQL, сопоставленное с данным прерыванием, так что *KiDispatchInterrupt* или *KiChainedDispatch* может перед вызовом ISR повысить IRQL до нужного уровня и вернуть его к исходному после завершения ISR. Этот двухэтапный процесс необходим из-за того, что при начальной диспетчеризации нельзя передать указатель (или какой-либо иной аргумент) объекту прерывания, так как она выполняется не программно, а аппаратно. В многопроцессорных системах ядро создает и инициализирует объект прерывания для каждого процессора, позволяя их локальным APIC принимать конкретные прерывания. На рис. 3-6 показана типичная схема обслуживания прерываний, сопоставленных с объектами прерываний.

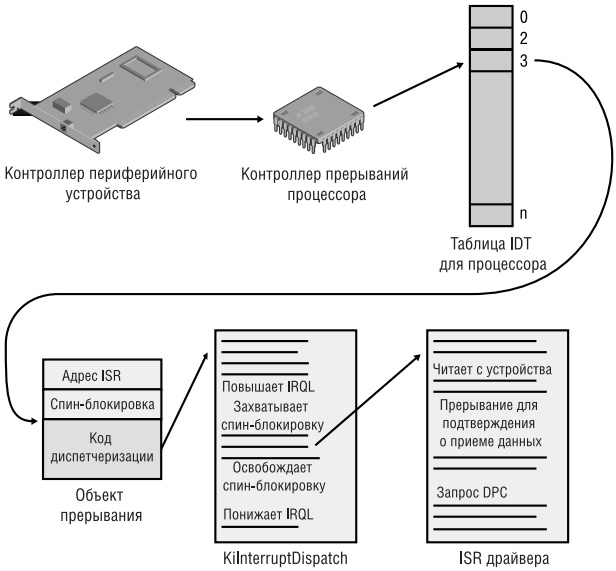


Рис. 3-6. *Типичная схема обслуживания прерываний*

ЭКСПЕРИМЕНТ: изучение внутреннего устройства прерываний

С помощью отладчика ядра вы можете просмотреть детальные сведения об объекте прерывания, в том числе его IRQL, адрес ISR и собственный код диспетчеризации прерывания (custom interrupt dispatching code). Во-первых, введите команду *!idt* и найдите запись со ссылкой на *I8042KeyboardInterruptService* — процедуру ISR для устройства «PS2-клавиатура»:

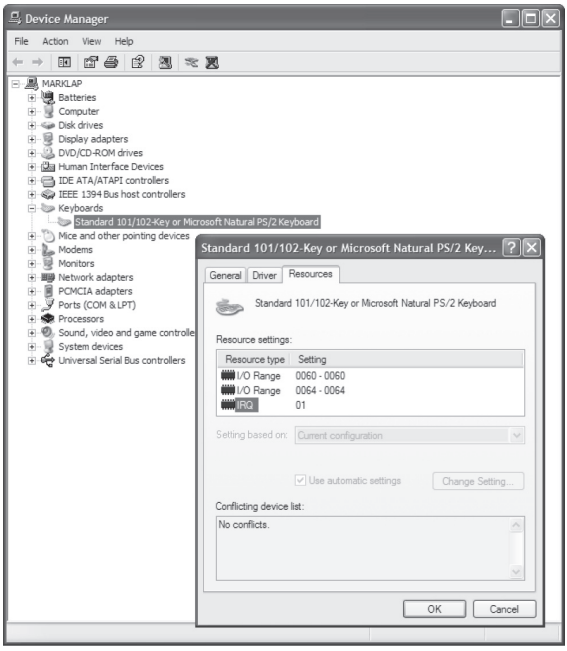
```
31:     8a39dc3c i8042prt!I8042KeyboardInterruptService
  (KINTERRUPT 8a39dc00)
```

Для просмотра содержимого объекта, сопоставленного с прерыванием, введите **dt nt!_kinterrupt**, указав адрес, следующий за KINTERRUPT:

```
kd> dt nt!_kinterrupt 8a39dc00
nt!_KINTERRUPT
   +0x000 Type             : 22
   +0x002 Size             : 484
   +0x004 InterruptListEntry : _LIST_ENTRY [ 0x8a39dc04 - 0x8a39dc04 ]
   +0x00c ServiceRoutine   : 0xba7e74a2
i8042prt!I8042KeyboardInterruptService+0
   +0x010 ServiceContext   : 0x8a067898
   +0x014 SpinLock         : 0
   +0x018 TickCount        : 0xffffffff
   +0x01c ActualLock       : 0x8a067958  -> 0
   +0x020 DispatchAddress  : 0x80531140     nt!KiInterruptDispatch+0
```

```
+0x024 Vector           : 0x31
+0x028 Irql             : 0x1a ''
+0x029 SynchronizeIrql  : 0x1a ''
+0x02a FloatingSave     : 0 ''
+0x02b Connected        : 0x1 ''
+0x02c Number           : 0 ''
+0x02d ShareVector      : 0 ''
+0x030 Mode             : 1 ( Latched )
+0x034 ServiceCount     : 0
+0x038 DispatchCount    : 0xffffffff
+0x03c DispatchCode     : [106] 0x56535554
```

В этом примере IRQL, назначенный Windows прерыванию, — 0x1a (или 26 в десятичной системе). Поскольку вывод получен на однопроцессорной x86-системе, IRQ равно 1 (IRQL в таких системах вычисляются путем вычитания IRQ из 27). Это можно проверить, открыв Device Manager (Диспетчер устройств) [на вкладке Hardware (Оборудование) в окне свойств системы)], найдя PS/2-клавиатуру и посмотрев назначенные ей ресурсы, как показано на следующей иллюстрации.



В многопроцессорных x86-системах IRQ назначается в основном случайным образом, а в x64- или IA64-системе вы увидите, что IRQ — это номер вектора прерываний [0x31 (49 в десятичной системе)], деленный на 16.

Адрес ISR для объекта прерывания хранится в поле ServiceRoutine (его и показывает *!idt* в своем выводе), а код прерывания, срабатыва-

см. след. стр.

ющий при появлении этого прерывания, — в массиве DispatchCode в конце объекта прерывания. Этот код программируется так, чтобы создавать фрейм ловушки в стеке и потом вызывать функцию, хранящуюся в поле DispatchAddress (в нашем примере это *KiInterruptDispatch*), с передачей ей указателя на объект прерывания.

Windows и обработка данных в реальном времени

К средам, предназначенным для работы в реальном времени, предъявляются либо жесткие, либо очень жесткие требования к максимальному времени реакции. Реакция системы реального времени, к которой предъявляются очень жесткие требования (например, системы управления атомной электростанцией), должна быть исключительно быстрой, иначе неизбежны катастрофы, опасные не только для оборудования, но и для жизни людей. Менее ответственные системы реального времени (например, система экономичного расхода топлива автомобиля) могут в какой-то мере отклоняться от этих требований, но их соблюдение все же желательно. В системах реального времени устройствами ввода служат датчики, а устройствами вывода — управляющие устройства. Проектировщик компьютерных систем реального времени должен знать величину максимально допустимого запаздывания между моментом генерации прерывания устройством ввода и ответом управляющего устройства, контролируемого драйвером. Анализ самых неблагоприятных вариантов должен учитывать как запаздывание операционной системы, так и запаздывание драйверов и приложений.

Поскольку проконтролировать расстановку приоритетов IRQ устройств операционной системой Windows нельзя, а пользовательские приложения выполняются лишь при IRQL уровня «passive», Windows не всегда подходит для обработки данных в реальном времени. Максимальное запаздывание определяется в конечном счете устройствами и драйверами системы, а не самой Windows. Этот фактор становится проблемой при использовании готового оборудования, имеющегося в продаже. Проектировщик может столкнуться с трудностями при определении максимального времени выполнения ISR или DPC драйвера готового устройства. Даже после тестирования он не сможет гарантировать, что запаздывание ни при каких обстоятельствах не превысит заданного предела. Более того, суммарное запаздывание системных DPC и ISR, как правило, существенно превосходит значения, приемлемые для чувствительных к задержкам систем.

Хотя ко многим типам встраиваемых систем (например, к принтерам и автомобильным компьютерам) предъявляются требования, как к системам реального времени, Windows XP Embedded не обладает соответствующими характеристиками. Это просто версия Windows XP, которая создана с использованием технологии, лицензированной Microsoft у компании VenturCom, и способна работать в системах с огра-

ниченными ресурсами. Так, для устройства без сетевых функций исключается вся функциональность Windows XP, связанная с поддержкой работы в сетях, включая средства управления сетью, драйверы стека протокола и сетевого адаптера.

Тем не менее третьи фирмы поставляют версии ядра реального времени для Windows. Их подход заключается в том, что они встраивают ядро реального времени в собственный HAL и выполняют Windows как задачу в операционной системе реального времени. Windows, выполняемая в таком виде, служит в качестве пользовательского интерфейса системы и имеет меньший приоритет по сравнению с задачами, ответственными за управление нужным устройством. Пример расширения ядра Windows реального времени можно увидеть на Web-сайте VenturCom *www.venturcom.com*.

Сопоставление ISR с конкретным уровнем прерывания называется *подключением объекта прерывания*, а разрыв связи между ISR и записью в IDT — *отключением*. Эти операции, выполняемые функциями ядра *IoConnectInterrupt* и *IoDisconnectInterrupt*, позволяют драйверу устройства «включать» ISR после своей загрузки и «отключать» ISR, если он не загружен.

Применение объекта прерывания для регистрации ISR позволяет драйверам устройств избегать прямого взаимодействия с контроллером прерываний (разным на разных процессорных архитектурах) и исключает необходимость детального знания IDT. Это дает возможность создавать переносимые драйверы устройств, поскольку благодаря такой функциональности ядра программировать драйверы устройств на ассемблере и учитывать в них особенности конкретных процессоров больше не нужно.

Использование объекта прерывания дает и другие преимущества: ядро может синхронизировать выполнение ISR с другими частями драйвера устройства, которые могут разделять данные с ISR. (Подробнее о том, как драйверы устройств реагируют на прерывания, см. главу 9.)

Более того, объекты прерывания позволяют ядру легко вызывать более одной ISR для любого уровня прерывания. Если несколько драйверов создают объекты прерывания и сопоставляют их с одной записью в IDT, то при прерывании на определенной линии диспетчер вызывает каждую из этих процедур (ISR). Такая функциональность позволяет ядру поддерживать конфигурации в виде цепочек, когда несколько устройств совместно используют одну линию прерывания. Когда одна из ISR объявляет диспетчеру о захвате прерывания, происходит разрыв цепочки. Если несколько устройств, разделяющих одну линию прерывания, одновременно запрашивают обслуживание, то устройства, не получившие подтверждения от своих ISR, будут вновь генерировать прерывания, как только диспетчер понизит IRQL. Связывание устройств в цепочку разрешается, только если все драйверы устройств, стремящиеся использовать одно и то же прерывание, сообщат ядру о своей способности разделять данное прерывание. Если они не в состоянии совместно использовать это прерывание, диспетчер Plug and Play пере-

назначит IRQ с учетом запросов каждого устройства. Если разделяемым является вектор прерываний, объект прерывания вызывает *KiChainedDispatch*, которая поочередно обращается к ISR каждого зарегистрированного объекта прерывания, пока один из них не сообщит, что прерывание вызвано им, или пока все они не будут выполнены. В одном из предыдущих примеров вывода *!idt* вектор 0x3b был подключен к нескольким объектам прерываний, связанным в цепочку (chained interrupt objects).

Программные прерывания

Хотя большинство прерываний генерируется аппаратно, ядро Windows тоже может генерировать прерывания — только они являются программными. Этот вид прерываний служит для решения многих задач, в том числе:

- инициации диспетчеризации потоков;
- обработки прерываний, не критичных по времени;
- обработки событий таймеров;
- асинхронного выполнения какой-либо процедуры в контексте конкретного потока;
- поддержки асинхронного ввода-вывода.

Эти задачи подробно рассматриваются ниже.

Прерывания DPC или диспетчеризации Когда дальнейшее выполнение потока невозможно, например из-за его завершения или перехода в ждущее состояние, ядро напрямую обращается к диспетчеру, чтобы вызвать немедленное переключение контекста. Однако иногда ядро обнаруживает, что перераспределение процессорного времени (rescheduling) должно произойти при выполнении глубоко вложенных уровней кода. В этой ситуации ядро запрашивает диспетчеризацию, но саму операцию откладывает до выполнения текущих действий. Такую задержку удобно организовать с помощью программного прерывания DPC (deferred procedure call).

При необходимости синхронизации доступа к разделяемым структурам ядра последнее всегда повышает IRQL процессора до уровня «DPC/dispatch» или выше. При этом дополнительные программные прерывания и диспетчеризация потоков запрещаются. Обнаружив необходимость в диспетчеризации, ядро генерирует прерывание уровня «DPC/dispatch». Но поскольку IRQL уже находится на этом уровне или выше, процессор откладывает обработку этого прерывания. Когда ядро завершает свои операции, оно определяет, что должно последовать снижение IRQL ниже уровня «DPC/dispatch», и проверяет, не ожидают ли выполнения отложенные прерывания диспетчеризации. Если да, IRQL понижается до уровня «DPC/dispatch», и эти отложенные прерывания обрабатываются. Активизация диспетчера потоков через программное прерывание — способ отложить диспетчеризацию до подходящего момента. Однако Windows использует программные прерывания для отложенного выполнения и других операций.

При этом IRQL ядро обрабатывает не только диспетчеризацию потоков, но и DPC. DPC — это функция, выполняющая системную задачу, менее критичную по времени в сравнении с текущей. Эти функции называются *отложенными* (deferred), так как не требуют немедленного выполнения.

DPC позволяют операционной системе генерировать прерывания и выполнять системные функции в режиме ядра. Ядро использует DPC для обработки прерываний по таймеру (и освобождения потоков, ждущих на таймерах), а также для перераспределения процессорного времени по истечении кванта времени, отведенного текущему потоку. Драйверы устройств используют DPC для выполнения запросов ввода-вывода. Для своевременного обслуживания аппаратных прерываний Windows — во взаимодействии с драйверами устройств — пытается удерживать текущий IRQL ниже IRQL устройств. Один из способов достижения этой цели заключается в следующем. ISR должна выполнять минимум действий по обслуживанию своего устройства, сохранять переменные данные о состоянии прерывания и откладывать передачу данных или выполнение других не столь критичных по времени операций, как DPC, до снижения IRQL к уровню «DPC/dispatch» (подробнее о DPC и системе ввода-вывода см. главу 9).

DPC представляется *DPC-объектом*, управляющим объектом ядра, невидимым программам пользовательского режима, но видимым драйверам и системному коду. Наиболее важной частью информации DPC-объекта является адрес системной функции, которую ядро должно вызвать для обработки прерывания DPC. DPC-процедуры, ожидающие выполнения, хранятся в управляемых ядром очередях (по одной на каждый процессор). Эти очереди называются *очередями DPC*. Запрашивая DPC, системный код вызывает ядро для инициализации DPC-объекта и помещает его в очередь DPC.

По умолчанию ядро помещает DPC-объекты в конец очереди DPC процессора, на котором был запрошен DPC (как правило, это процессор, на котором выполняется ISR). Однако драйвер устройства может изменить это, указав приоритет DPC (низкий, средний или высокий; по умолчанию — средний) или направив DPC конкретному процессору. DPC, направленный конкретному процессору, называется *целевым DPC* (targeted DPC). Если у DPC низкий или средний приоритет, ядро помещает DPC-объект в конец очереди, а если у DPC высокий приоритет, то — в начало.

Когда IRQL процессора вот-вот понизится с уровня «DPC/dispatch» или более высокого до уровня «APC» или «passive», ядро переходит к обработке всех DPC. Windows оставляет IRQL на уровне «DPC/dispatch» и извлекает все DPC-объекты из очереди данного процессора (т. е. ядро опустошает очередь), поочередно вызывая каждую DPC-функцию. Ядро разрешает уменьшить IRQL ниже уровня «DPC/dispatch» для продолжения выполнения обычных потоков только после опустошения очереди. Схема обработки DPC показана на рис. 3-7.

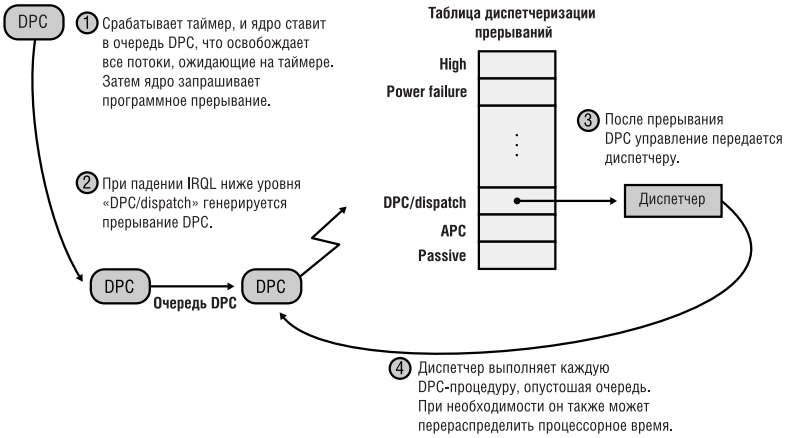


Рис. 3-7. *Обработка DPC*

Приоритеты DPC могут влиять на поведение системы и иным способом. Обычно ядро начинает опустошение очереди DPC с прерывания уровня «DPC/dispatch». Такое прерывание генерируется ядром, только если DPC направлен на процессор, на котором выполняется ISR, и DPC имеет средний или высокий приоритет. Если у DPC низкий приоритет, ядро генерирует прерывание, только если число незавершенных запросов DPC превышает пороговое значение или если число DPC, запрошенных на процессоре за установленный период, невелико. Если DPC направлен другому процессору (не тому, на котором выполняется ISR) и его приоритет высокий, ядро немедленно посылает ему диспетчерское IPI, сигнализируя целевому процессору о необходимости опустошения его очереди DPC. Если приоритет DPC средний или низкий, для появления прерывания «DPC/dispatch» число DPC в очереди целевого процессора должно превышать пороговое значение. Системный поток простоя также опустошает очередь DPC процессора, на котором он выполняется. Хотя уровни приоритета и направление DPC являются довольно гибкими средствами, у драйверов устройств редко возникает необходимость в изменении заданного по умолчанию поведения своих DPC-объектов. В таблице 3-1 даются сведения о ситуациях, в которых начинается опустошение очереди DPC.

Таблица 3-1. *Правила генерации прерываний DPC*

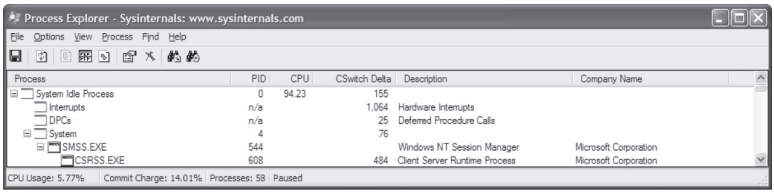
Приоритет DPC	DPC, направленный на процессор ISR	DPC, направленный на другой процессор
Низкий	Длина очереди DPC превышает пороговое значение, или частота запросов DPC ниже минимальной	Длина очереди DPC превышает пороговое значение, или система не занята
Средний	Всегда	Длина очереди DPC превышает пороговое значение, или система не занята
Высокий	Всегда	Всегда

Поскольку потоки пользовательского режима выполняются при низком IRQL, вероятность того, что DPC прервет выполнение обычного пользовательского потока, довольно велика. DPC-процедуры выполняются независимо от того, какой поток работает в настоящий момент. Это означает, что выполняемая DPC-процедура не в состоянии предугадать текущий размер спроецированного адресного пространства процесса. DPC-процедуры могут вызывать функции ядра, но не могут обращаться к системным сервисам, генерировать ошибки страницы, создавать или ждать объекты диспетчера. Однако они способны получать доступ к неподкачиваемым областям системной памяти, поскольку системное адресное пространство всегда спроецировано независимо от того, что представляет собой текущий процесс.

DPC используются не только драйверами, но и ядром. Ядро чаще всего применяет DPC для обработки ситуации, когда истекает выделенный квант времени. При каждом такте системного таймера генерируется прерывание с IRQL-уровнем «clock». Обработчик прерываний таймера (выполняемый при IRQL, равном «clock») обновляет системное время и уменьшает значение счетчика, отслеживающего время выполнения текущего потока. Когда значение счетчика обнуляется, квант времени, отведенный потоку, заканчивается, и ядру может понадобиться перераспределить процессорное время — эта задача имеет более низкий приоритет и должна выполняться при IRQL, равном «DPC/dispatch». Обработчик прерываний таймера ставит DPC в очередь, чтобы инициировать диспетчеризацию потоков, после чего завершает свою работу и понижает IRQL процессора. Поскольку приоритет прерываний DPC ниже, чем аппаратных, перед генерацией прерывания DPC сначала обрабатываются все аппаратные прерывания, возникающие до завершения обработки прерывания таймера.

ЭКСПЕРИМЕНТ: мониторинг активности прерываний и DPC

Process Explorer позволяет вести мониторинг активности прерываний и DPC, добавив столбец Context Switch Delta и наблюдая за процессами Interrupt и DPC. Это не настоящие процессы, они показываются как процессы просто для удобства и не вызывают переключений контекста. Счетчик переключений контекста в Process Explorer для этих псевдопроцессов отражает число возникновений каждого из них в течение предыдущего интервала обновления (refresh interval). Вы можете имитировать активность прерываний и DPC, быстро перемещая курсор мыши по экрану.



см. след. стр.

Вы также можете проследить выполнение конкретных процедур обслуживания прерываний (ISR) и отложенных вызовов процедур (DPC), используя встроенную поддержку трассировки событий (подробнее об этом будет рассказано позже) в Windows XP Service Pack 2 или Windows Server 2003 Service Pack 1 (и выше).

1. Инициируйте захват событий, введя команду:

```
tracelog -start -f kernel.etl -b 64 -UsePerfCounter
-eflag 8 0x307 0x4084 0 0 0 0 0 0
```

2. Остановите захват событий, введя:

```
tracelog -stop
```

3. Создайте отчеты по захваченным событиям, набрав команду:

```
tracerpt kernel.etl -df -o -report
```

Это приведет к генерации двух файлов: workload.txt и dumpfile.csv.

4. Откройте workload.txt и вы увидите сводные сведения о времени, проведенном драйверами каждого типа в ISR- и DPC-процедурах.
5. Откройте файл dumpfile.csv, созданный на этапе 3; найдите строки, где во втором значении встречается «DPC» или «ISR». Например, следующие три строки из dumpfile.csv показывают DPC таймера, DPC и ISR:

```
PerfInfo,   TimerDPC, 0xFFFFFFFF,   127383953645422825,
      0,          0, 127383953645421500, 0xFB03A385, 0, 0
PerfInfo,         DPC, 0xFFFFFFFF,   127383953645424040,
      0,          0, 127383953645421394, 0x804DC87D, 0, 0
PerfInfo,         ISR, 0xFFFFFFFF,   127383953645470903,
      0,          0, 127383953645468696, 0xFB48D5E0, 0, 0, 0
```

Введя команду *ln* в отладчике ядра и указав начальный адрес из каждой записи события (восьмое значение в каждой строке), вы увидите имя функции, выполнявшей DPC или ISR:

```
lkd> ln 0xFB03A385
(fb03a385)   rdbss!RxTimerDispatch   |  (fb03a41e)
   rdbss!RxpWorkerThreadDispatcher
lkd> ln 0x804DC87D
(804dc87d)   nt!KiTimerExpiration   |  (804dc93b)
   nt!KeSetTimerEx
lkd> ln 0xFB48D5E0
(fb48d5e0)   atapi!IdePortInterrupt   |  (fb48d622)
   atapi!IdeCheckEmptyChannel
```

Первый адрес относится к DPC, вызванному срабатыванием таймера, который был поставлен в очередь клиентским драйвером редиректора файловой системы (file system redirector client driver). Второй относится к DPC, вызванному срабатыванием универсального таймера

(generic timer). Наконец, третий — это адрес ISR для порт-драйвера ATAPI. Более подробные сведения см. по ссылке *http://www.microsoft.com/whdc/driver/perform/mmdrv.mspx*.

Прерывания APC APC (asynchronous procedure call) позволяет выполнять пользовательские программы и системный код в контексте конкретного пользовательского потока (а значит, и в адресном пространстве конкретного процесса). Поскольку для выполнения в контексте конкретного потока APC ставятся в очередь и выполняются при IRQL ниже «DPC/dispatch», на их работу не налагаются ограничения, свойственные DPC. APC-процедура может обращаться к ресурсам (объектам), ждать освобождения описателей объектов, генерировать ошибки страниц и вызывать системные сервисы.

APC описывается управляющим объектом ядра — *APC-объектом*. APC, ждущие выполнения, находятся в *очереди APC* (APC queue), управляемой ядром. Очереди APC — в отличие от общесистемной очереди DPC — специфичны для конкретного потока, так как у каждого потока своя очередь APC. При запросе на постановку APC в очередь ядро помещает его (APC) в очередь того потока, который будет выполнять APC-процедуру. Далее ядро генерирует программное прерывание с уровнем APC, и поток, когда он в конечном счете начинает выполняться, обрабатывает APC.

APC бывают двух видов: режима ядра и пользовательского режима. APC режима ядра для выполнения в контексте целевого потока не нужно «разрешение» со стороны этого потока, тогда как для APC пользовательского режима это обязательно. APC режима ядра прерывает поток и выполняет процедуру без вмешательства или согласия потока. APC режима ядра тоже бывают двух типов: обычные (normal) и специальные (special). Поток может отключить все APC режима ядра, повысив IRQL до уровня APC_LEVEL или вызвав *KeEnterGuardedRegion*, которая впервые появилась в Windows Server 2003. *KeEnterGuardedRegion* отключает доставку APC, устанавливая поле SpecialApcDisable в структуре KTHREAD вызвавшего потока (об этой структуре см. главу 6). Поток также может отключить только обычные APC режима ядра вызовом *KeEnterCriticalRegion*, которая устанавливает поле KernelApcDisable в структуре KTHREAD потока.

Исполнительная система использует APC режима ядра для тех задач операционной системы, которые нужно выполнить в адресном пространстве (контексте) конкретного потока. Так, через специальные APC режима ядра она может указать потоку прекратить выполнение системного сервиса, допускающего прерывание, или записать результаты операции асинхронного ввода-вывода в адресное пространство этого потока. Подсистемы окружения используют такие APC, чтобы приостановить поток или завершить себя, а также чтобы получить или установить контекст пользовательского потока. Подсистема POSIX эмулирует через APC режима ядра передачу POSIX-сигналов процессам POSIX.

Драйверы устройств также применяют APC режима ядра. Например, если инициирована операция ввода-вывода и поток переходит в состояние ожи-

дания, к процессору может быть подключен другой поток из другого процесса. По завершении передачи данных устройством система ввода-вывода должна как-то вернуться в контекст потока, инициировавшего эту операцию ввода-вывода, чтобы он мог скопировать ее результаты в буфер в адресном пространстве своего процесса. Система ввода-вывода использует для выполнения подобных действий специальные APC режима ядра (применение APC в системе ввода-вывода подробно рассматривается в главе 9).

Некоторые Windows-функции вроде *ReadFileEx*, *WriteFileEx* и *QueueUserAPC* вызывают APC пользовательского режима. Так, функции *ReadFileEx* и *WriteFileEx* позволяют вызывающей программе указать процедуру завершения ввода-вывода (completion procedure), которая будет вызвана по окончании операции ввода-вывода. Процедура завершения ввода-вывода реализуется помещением APC в очередь потока, выдавшего запрос на ввод-вывод. Однако обратный вызов процедуры завершения не обязательно происходит в момент постановки APC в очередь, поскольку APC пользовательского режима передаются потоку, только если он находится в *состоянии тревожного ожидания* (alertable wait state). Поток может перейти в такое состояние, вызвав одну из Windows-функций: либо *WaitForMultipleObjectsEx*, либо *SleepEx*. В обоих случаях, как только в очереди появится APC пользовательского режима, ядро прервет поток, передаст управление APC-процедуре и возобновит его выполнение лишь после завершения APC-процедуры. В отличие от APC режима ядра, которые выполняются на уровне «APC», APC пользовательского режима выполняются на уровне «passive».

Появление APC может переупорядочить очереди ожидания — списки, определяющие, какие потоки ждут, в каком порядке и на каких объектах (см. раздел по синхронизации далее в этой главе). Если в момент появления APC поток находится в состоянии ожидания, то после обработки APC-процедуры поток возвращается в состояние ожидания, но перемещается в конец списка потоков, ждущих те же объекты.

Диспетчеризация исключений

В отличие от прерываний, которые могут возникать в любой момент, исключения являются прямым следствием действий выполняемой программы. Windows вводит понятие *структурной обработки исключений* (structured exception handling, SEH), позволяющей приложениям получать управление при возникновении исключений. При этом приложение может исправить ситуацию, которая привела к исключению, провести раскрутку стека (завершив таким образом выполнение подпрограммы, вызвавшей исключение) или уведомить систему о том, что данное исключение ему не известно, и тогда система продолжит поиск подходящего обработчика для данного исключения. В этом разделе мы исходим из того, что вы знакомы с базовыми концепциями структурной обработки исключений Windows; в ином случае сначала прочитайте соответствующую часть справочной документации Windows API из Platform SDK или главы 23–25 из четвертого издания книги Джеффри Рихтера «Windows для профессионалов». Учтите: хотя обработка

исключений возможна через расширения языка программирования (например, с помощью конструкции *__try* в Microsoft Visual C++), она является системным механизмом и поэтому не зависит от конкретного языка.

В системах типа x86 все исключения имеют предопределенные номера прерываний, прямо соответствующие записям в IDT, ссылающимся на обработчики ловушек конкретных исключений. В таблице 3-2 перечислены исключения, определенные для систем типа x86, с указанием номеров прерываний. Как уже говорилось, первая часть IDT используется для исключений, а аппаратные прерывания располагаются за ней.

Таблица 3-2. *Исключения в системах типа x86 и соответствующие им номера прерываний*

Номер прерывания	Исключение
0	Divide Error (ошибка деления)
1	DEBUG TRAP (ловушка отладки)
2	NMI/NPX Error (ошибка NMI/NPX)
3	Breakpoint (точка прерывания)
4	Overflow (переполнение)
5	BOUND/Print Screen
6	Invalid Opcode (неправильный код операции)
7	NPX Not Available (NPX недоступен)
8	Double Exception (двойное исключение)
9	NPX Segment Overrun (выход за пределы сегмента NPX)
A	Invalid Task State Segment (TSS) (неправильный TSS)
B	Segment Not Present (сегмент отсутствует)
C	Stack Fault (ошибка стека)
D	General Protection (ошибка общей защиты)
E	Page Fault (ошибка страницы)
F	Зарезервировано Intel
10	Floating Point (ошибка в операции с плавающей точкой)
11	Alignment Check (ошибка контроля выравнивания)

Все исключения, кроме достаточно простых, которые могут быть разрешены обработчиком ловушек, обслуживаются модулем ядра — *диспетчером исключений* (exception dispatcher). Его задача заключается в поиске обработчика, способного «справиться» с данным исключением. Примерами независимых от архитектуры исключений могут служить нарушения доступа к памяти, целочисленное деление на нуль, переполнение целых чисел, исключения при операциях с плавающей точкой и точки прерывания отладчика. Полный список независимых от архитектуры исключений см. в справочной документации Windows API.

Ядро перехватывает и обрабатывает некоторые из этих исключений прозрачно для пользовательских программ. Так, если при выполнении отлаживаемой программы встретилась точка прерывания, генерируется исключе-

ние, обрабатываемое ядром за счет вызова отладчика. Ряд исключений ядро обрабатывает, просто возвращая код неудачной операции.

Определенные исключения могут передаваться в неизменном виде пользовательским процессам. Например, при ошибке доступа к памяти или при переполнении в ходе арифметической операции генерируется исключение, не обрабатываемое операционной системой. Для обработки этих исключений подсистема окружения может устанавливать *обработчики исключений на основе SEH-фрейма* (далее для краткости — обработчик SEH-фрейма). Этим термином обозначается обработчик исключения, сопоставленный с вызовом конкретной процедуры. При активизации такой процедуры в стек заталкивается *стековый фрейм*, представляющий вызов этой процедуры. Со стековым фреймом можно сопоставить один или несколько обработчиков исключений, каждый из которых защищает определенный блок кода исходной программы. При возникновении исключения ядро ищет обработчик, сопоставленный с текущим стековым фреймом. Если его нет, ядро ищет обработчик, сопоставленный с предыдущим стековым фреймом, — и так до тех пор, пока не будет найден подходящий обработчик. Если найти обработчик исключения не удалось, ядро вызывает собственные обработчики по умолчанию.

Когда происходит исключение (аппаратное или программное), цепочка событий начинается в ядре. Процессор передает управление обработчику ловушки в ядре, который создает фрейм ловушки по аналогии с тем, как это происходит при прерывании. Фрейм ловушки позволяет системе после обработки исключения возобновить работу с той точки, где она была прервана. Обработчик ловушки также создает запись исключения, содержащую сведения о ее причине и другую сопутствующую информацию.

Если исключение возникает в режиме ядра, то для его обработки диспетчер исключений просто вызывает процедуру поиска подходящего обработчика SEH-фрейма. Поскольку необработанные исключения режима ядра были бы фатальными ошибками операционной системы, диспетчер всегда находит какой-нибудь обработчик.

Если исключение возникает в пользовательском режиме, диспетчер исключений предпринимает более сложные действия. Как поясняется в главе 6, подсистема Windows предусматривает порт отладчика (debugger port) и порт исключений (exception port) для приема уведомлений об исключениях пользовательского режима в Windows-процессах. Они применяются ядром при обработке исключений по умолчанию, как показано на рис. 3-8.

Точки прерывания в отлаживаемой программе являются распространенной причиной исключений. Поэтому диспетчер исключений первым делом проверяет, подключен ли к процессу, вызвавшему исключение, отладчик. Если подключен и системой является Windows 2000, диспетчер исключений посылает отладчику через LPC первое предупреждение. (Это уведомление на самом деле сначала поступает диспетчеру сеансов, а тот пересылает его соответствующему отладчику.) В Windows XP и Windows Server 2003 диспетчер исключений посылает сообщение объекта отладчика (debugger object mes-

sage) *объекту отладки* (debug object), сопоставленному с процессом (который внутри системы рассматривается как порт).

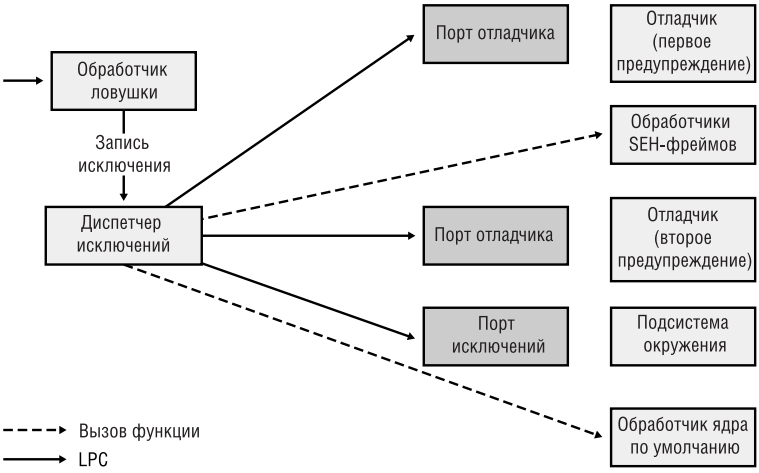


Рис. 3-8. *Диспетчеризация исключений*

Если к процессу не подключен отладчик или если отладчик не в состоянии обработать данное исключение, диспетчер исключений переключается в пользовательский режим, копирует фрейм ловушки в пользовательский стек, имеющий формат структуры данных CONTEXT (документирована в Platform SDK), и вызывает процедуру поиска обработчика SEH-фрейма. Если поиск не дал результатов, диспетчер возвращается в режим ядра и снова вызывает отладчик, чтобы пользователь мог продолжить отладку. При этом посылается второе (и последнее) предупреждение.

Если отладчик не запущен и обработчики SEH-фреймов не найдены, ядро посылает сообщение в порт исключений, сопоставленный с процессом потока. Этот порт (если таковой есть) регистрируется подсистемой окружения, контролирующей данный поток. Порт исключений дает возможность подсистеме окружения (прослушивающей этот порт) транслировать исходное исключение в уведомление или исключение, специфичное для ее окружения. CSRSS (Client/Server Run-Time Subsystem) просто выводит окно сообщения, уведомляющее пользователя о сбое, и завершает процесс. Когда подсистема POSIX получает от ядра сообщение о том, что один из потоков вызвал исключение, эта подсистема посылает вызвавшему исключение потоку сигнал в стиле POSIX. Но, если ядро уже дошло до этого этапа в обработке исключения, а подсистема не способна обработать данное исключение, выполняется обработчик ядра по умолчанию, просто завершающий процесс, поток которого вызвал исключение.

Необработанные исключения

На вершине стека любого Windows-потока объявляется обработчик, имеющий дело с необработанными исключениями. За объявление отвечает внут-

ренняя Windows-функция *start-of-process* или *start-of-thread*. Функция *start-of-process* срабатывает в момент начала выполнения первого потока процесса. Она вызывает главную точку входа в образе. Функция *start-of-thread* выполняется при создании дополнительных потоков в процессе и вызывает стартовую процедуру, указанную в вызове *CreateThread*.

ЭКСПЕРИМЕНТ: определение истинного стартового адреса Windows-потоков

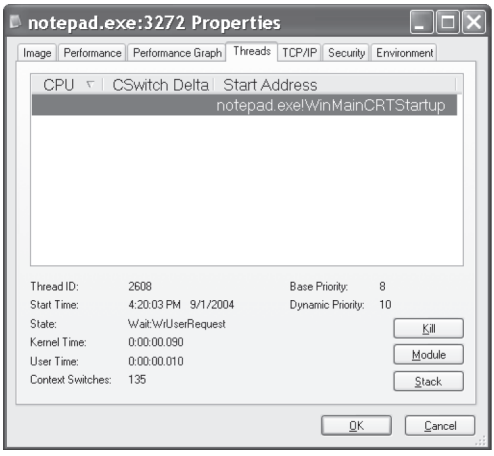
Тот факт, что выполнение каждого Windows-потока начинается с системной (а не пользовательской) функции, объясняет, почему у каждого Windows-процесса стартовый адрес нулевого потока одинаков (как и стартовые адреса вторичных потоков). Стартовый адрес нулевого потока Windows-процессов соответствует Windows-функции *start-of-process*, а стартовые адреса остальных потоков являются адресом Windows-функции *start-of-thread*. Для определения адреса пользовательской функции применим утилиту Tlist из Windows Support Tools. Для получения детальной информации о процессе наберите **tlist *имя_процесса*** или **tlist *идентификатор_процесса***. Например, сравним стартовый адрес процесса Windows Explorer, сообщаемый утилитой Pstat (из Platform SDK), и стартовый адрес Tlist.

```
C:\> pstat
...
pid:3f8 pri: 8 Hnd:  329 Pf:  80043 Ws:   4620K explorer.exe
 tid pri Ctx Swtch StrtAddr   User Time Kernel Time  State
  7c   9     16442 77E878C1 0:00:01.241 0:00:01.251 Wait:UserRequest
 42c  11    157888 77E92C50 0:00:07.110 0:00:34.309 Wait:UserRequest
 44c   8      6357 77E92C50 0:00:00.070 0:00:00.140 Wait:UserRequest
 1cc   8      3318 77E92C50 0:00:00.030 0:00:00.070 Wait:DelayExecution
 ...

C:\> tlist explorer
 1016 explorer.exe      Program Manager
   CWD:     C:\
   CmdLine: Explorer.exe
   VirtualSize:    25348 KB   PeakVirtualSize:    31052 KB
   WorkingSetSize:  1804 KB   PeakWorkingSetSize:  3276 KB
   NumberOfThreads: 4
    149 Win32StartAddr:0x01009dbd LastErr:0x0000007e State:Waiting
     86 Win32StartAddr:0x77c5d4a5 LastErr:0x00000000 State:Waiting
     62 Win32StartAddr:0x00000977 LastErr:0x00000000 State:Waiting
    179 Win32StartAddr:0x0100d8d4 LastErr:0x00000002 State:Waiting
    ...
```

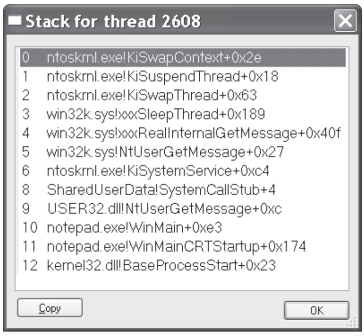
Стартовый адрес нулевого потока, сообщаемый Pstat, соответствует внутренней Windows-функции *start-of-process*, а стартовые адреса потоков 1–3 указывают адреса внутренних Windows-функций *start-of-thread*. С другой стороны, Tlist показывает стартовый адрес пользова-

тельской функции, вызываемой внутренней стартовой Windows-функцией.

Поскольку большинство потоков в Windows-процессах начинается в одной из системных функций-оболочек, Process Explorer, показывая стартовые адреса потоков в процессе, пропускает фрейм начального вызова, представляющий функцию-оболочку, и вместо этого отображает второй фрейм в стеке. Например, обратите внимание на стартовый адрес потока в процессе, выполняющем Notepad.exe.



Process Explorer не выводит всю иерархию вызовов при отображении стека вызовов. Вот что вы получите, щелкнув кнопку Stack.



В строке 12 на этой иллюстрации показан первый фрейм в стеке — начало процесса-оболочки. Второй фрейм (строка 11) является основной точкой входа в Notepad.exe.

Базовый код внутренних стартовых функций выглядит так:

```
void Win32StartOfProcess(
   LPTHREAD_START_ROUTINE lpStartAddr, LPVOID lpvThreadParm){

   __try {
```

```
      DWORD dwThreadExitCode = lpStartAddr(lpvThreadParm);
      ExitThread(dwThreadExitCode);

   } __except(UnhandledExceptionFilter(
         GetExceptionInformation())) {

      ExitProcess(GetExceptionCode());
   }
}
```

Заметьте: если при выполнении потока возникает исключение, не обрабатываемое этим потоком, вызывается Windows-фильтр необработанных исключений. Эта функция реализует поведение системы, когда та обнаруживает необработанное исключение. Поведение зависит от содержимого раздела реестра HKLM\Software\Microsoft\Windows NT\CurrentVersion\AeDebug. В нем есть два важных параметра: Auto и Debugger. Auto сообщает фильтру необработанных исключений, надо ли автоматически запускать отладчик или спросить у пользователя, что делать. По умолчанию этому параметру присваивается 1, что подразумевает автоматический запуск отладчика. Однако после установки средств разработки вроде Visual Studio его значение меняется на 0. Параметр Debugger является строкой, которая указывает путь к исполняемому файлу отладчика, который следует запускать при появлении необработанного исключения.

Отладчик по умолчанию — \Windows\System32\Drwtsn32.exe (Dr. Watson), который на самом деле является не отладчиком, а утилитой, сохраняющей сведения о рухнувшем приложении в файле журнала (Drwtsn32.log) и обрабатывающей файл аварийного дампа (User.dmp). Оба этих файла по умолчанию помещаются в папку \Documents And Settings\All Users\ Documents\DrWatson. Для просмотра или изменения конфигурации утилиту Dr. Watson можно запустить в интерактивном режиме; при этом выводится окно с текущими параметрами (пример для Windows 2000 показан на рис. 3-9).

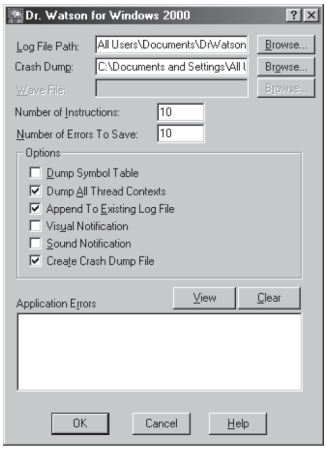


Рис. 3-9. *Параметры Dr. Watson (Доктор Ватсон) по умолчанию*

Файл журнала содержит такую базовую информацию, как код исключения, имя рухнувшего образа, список загруженных DLL, а также содержимое стека и последовательность команд потока, вызвавшая исключение. Более подробные сведения о содержимом этого файла можно получить, запустив Dr. Watson и щелкнув кнопку Help (Справка) в его окне.

В файл аварийного дампа записывается содержимое закрытых страниц процесса на момент возникновения исключения (но страницы кода из EXE- и DLL-модулей не включаются). Этот файл можно открыть с помощью WinDbg — Windows-отладчика, поставляемого с пакетом Debugging Tools или с Visual Studio 2003 и выше). Файл аварийного дампа перезаписывается при каждом крахе процесса. Поэтому, если его предварительно не скопировать или не переименовать, в нем будет содержаться информация лишь о последнем крахе процесса.

В Windows 2000 Professional визуальное уведомление включено по умолчанию. Окно сообщения, представленное на рис. 3-10, выводится Dr. Watson после того, как он сгенерирует аварийный дамп и запишет информацию в свой файл журнала.

Рис. 3-10. *Сообщение об ошибке от Dr. Watson (Windows 2000)*

Процесс Dr. Watson остается до тех пор, пока не будет закрыто это окно, и именно поэтому в Windows 2000 Server визуальное уведомление по умолчанию отключено. Дело вот в чем. Обычно сервер находится в отдельной комнате и возле него никто не сидит. Если на сервере рушится какое-то приложение, то подобное окно просто некому закрыть. По этой причине серверные приложения должны регистрировать ошибки в журнале событий Windows.

В Windows 2000, если параметр Auto установлен в 0, отображается окно, приведенное на рис. 3-11.

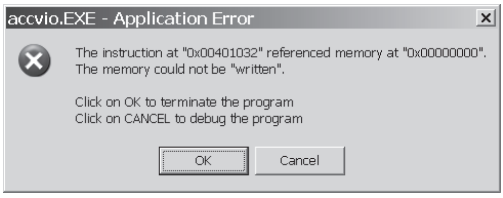


Рис. 3-11. *Сообщение о необработанном исключении (Windows 2000)*

После щелчка кнопки OK процесс завершается. А если вы нажимаете кнопку Cancel, запускается системный отладчик (заданный параметром Debugger в реестре).

ЭКСПЕРИМЕНТ: необработанные исключения

Чтобы увидеть образец файла журнала Dr. Watson, запустите программу Accvio.exe, которую можно скачать по ссылке *www.sysinternals.com/windowsinternals.shtml*. Эта программа вызовет нарушение доступа (ошибку защиты памяти) при попытке записи по нулевому адресу, всегда недействительному для Windows-процессов (см. таблицу 7-6 в главе 7).

1. Запустите Registry Editor (Редактор реестра) и найдите раздел HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug.
2. Если значение параметра Debugger равно «drwtsn32 -p %ld -e %ld -g», ваша система настроена на использование Dr. Watson в качестве отладчика по умолчанию. Переходите в п. 4.
3. Если в параметре Debugger не указан Drwtsn32.exe, вы все равно можете протестировать Dr. Watson, временно установив его, а затем восстановив исходные параметры своего отладчика:
   - сохраните где-нибудь текущее значение параметра (например, в файле Notepad или в буфере обмена);
   - выберите из меню Start (Пуск) команду Run (Выполнить) и введите команду **drwtsn32 -i** (чтобы инициализировать параметр Debugger для запуска Dr. Watson).
4. Запустите тестовую программу Accvio.exe.
5. Вы должны увидеть одно из окон, описанных ранее (в зависимости от версии Windows, в которой вы работаете).
6. Если вы используете параметры Dr. Watson по умолчанию, то теперь сможете изучить файл журнала и файл дампа в каталоге файлов дампов. Для просмотра параметров Dr. Watson, запустите *drwtsn32* без аргументов. (Выберите из меню Start команду Run и введите **drwtsn32**.)
7. В качестве альтернативы щелкните последнюю запись в списке Application Errors (Ошибки приложения) и нажмите кнопку View (Показать) — будет выведена часть файла журнала Dr. Watson, содержащая сведения о нарушении доступа, вызванном Accvio.exe. Если вас интересуют детали формата файла журнала, щелкните кнопку Help (Справка) и выберите раздел Dr. Watson Log File Overview (Обзор файла журнала доктора Ватсона).
8. Если Dr. Watson не был отладчиком по умолчанию, восстановите исходное значение, сохраненное в п. 1.

Проведите еще один эксперимент: попробуйте перенастроить параметр Debugger на другую программу, например Notepad.exe (Блокнот) или Sol.exe (Solitaire). Снова запустите Accvio.exe. Обратите внимание, что запускается любая программа, указанная в параметре Debugger. То есть система не проверяет, действительно ли указанная в этом параметре программа является отладчиком. Обязательно восстановите исходные значения параметров реестра (введите **drwtsn32 -i**).

Windows-поддержка отчетов об ошибках

В Windows XP и Windows Server 2003 появился новый, более изощренный механизм отчетов об ошибках, называемый Windows Error Reporting. Он автоматизирует передачу информации о крахе как в пользовательском режиме, так и в режиме ядра. (Как применить этот механизм для получения сведений о крахе системы, см. главу 14.)

Windows Error Reporting можно настроить, последовательно выбрав My Computer (Мой компьютер), Properties (Свойства), Advanced (Дополнительно) и Error Reporting (Отчет об ошибках) (на экране появится диалоговое окно, показанное на рис. 3-12); то же самое можно сделать через параметры локальной или доменной политики группы, которые хранятся в разделе реестра HKLM\Software\Microsoft\PCHealth\ErrorReporting.

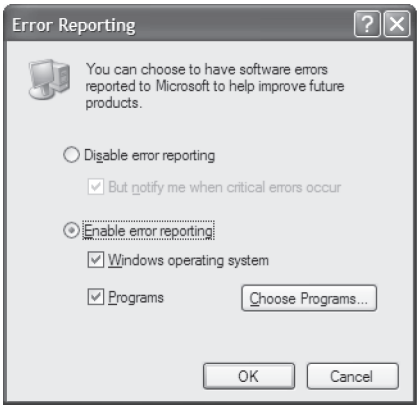


Рис. 3-12. *Диалоговое окно настройки отчетов об ошибках*

Перехватив необработанное исключение (об этом шла речь в предыдущем разделе), фильтр необработанных исключений выполняет начальную проверку, чтобы решить, надо ли запустить механизм Windows Error Reporting. Если параметр реестра HKLM\SOFTWARE\Microsoft\Windows NT\Current Version\AeDebug\Auto установлен в 0 или строка Debugger содержит текст «Drwtsn32», фильтр необработанных исключений загружает в аварийный процесс библиотеку \Windows\System32\Faultrep.dll и вызывает ее функцию *ReportFault*. Эта функция проверяет конфигурацию механизма отчетов об ошибках, которая хранится в разделе HKLM\Software\Microsoft\PCHealth\ErrorReporting, и определяет, следует ли формировать отчет для данного процесса и, если да, то как. В обычном случае *ReportFault* создает процесс, выполняющий \Windows\System32\Dwwin.exe, который выводит окно, где пользователь уведомляется о крахе процесса и где ему предоставляется возможность передать отчет об ошибках в Microsoft (рис. 3-13).

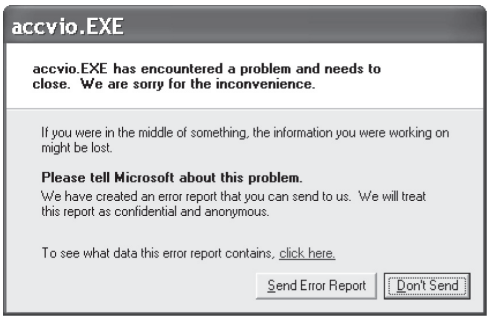


Рис. 3-13. *Диалоговое окно, выводимое Windows Error Reporting*

При щелчке кнопки Send Error Report (Послать отчет), отчет об ошибках (минидамп и текстовый файл с детальными сведениями о номерах версий DLL, загруженных в рухнувший процесс) передается на онлайновый сервер анализа аварийных ситуаций, Watson.Microsoft.com. (В отличие от краха системы в режиме ядра здесь нет возможности найти какое-либо решение на момент отправки отчета.) Затем фильтр необработанных исключений создает процесс для запуска отладчика (обычно Drwtsn32.exe), который по умолчанию создает свой файл дампа и запись в журнале. В отличие от Windows 2000 этот файл содержит не полный дамп, а минидамп. Поэтому в ситуации, где для отладки рухнувшего приложения нужен полный дамп памяти процесса, вы можете изменить конфигурацию Dr. Watson, запустив его без аргументов командной строки, как было описано в предыдущем разделе.

В средах, где системы не подключены к Интернету или где администратор хочет контролировать, какие именно отчеты об ошибках посылаются в Microsoft, эти отчеты можно передавать на внутренний файл-сервер. Microsoft предоставляет опытным заказчикам утилиту Corporate Error Reporting, которая понимает структуру каталогов, создаваемую Windows Error Reporting, и позволяет администратору задавать условия, при которых отчеты формируются и передаются в Microsoft. (Подробности см. по ссылке *http://www.microsoft.com/resources/satech/cer*.)

Диспетчеризация системных сервисов

Как показано на рис. 3-1, обработчики ловушек ядра обслуживают прерывания, исключения и вызовы системных сервисов. Из предыдущих разделов вы знаете, как проводится обработка прерываний и исключений. Здесь будет рассказано о вызовах системных сервисов. Диспетчеризация системных сервисов начинается с выполнения инструкции, закрепленной за такой диспетчеризацией. Эта инструкция зависит от процессора, на котором работает Windows.

Диспетчеризация 32-разрядных системных сервисов

На процессорах x86 до Pentium II использовалась инструкция *int 0x2e* (десятичное значение 46). В результате выполнения этой инструкции срабатывает ловушка, и Windows заносит в запись IDT под номером 46 указатель на

диспетчер системных сервисов (см. таблицу 3-1). Эта ловушка заставляет выполняемый поток переключиться в режим ядра и войти в диспетчер системных сервисов. Номер запрошенного системного сервиса указывается числовым аргументом, переданным в регистр процессора EAX. Содержимое регистра EBX указывает на список параметров, передаваемый системному сервису вызывающей программой.

На x86-процессорах Pentium II и выше Windows использует инструкцию *sysenter*, которую Intel специально определил для быстрой диспетчеризации системных сервисов. Для поддержки этой инструкции Windows сохраняет на этапе загрузки адрес процедуры ядра — диспетчера системных сервисов в регистре, сопоставленном с данной инструкцией. Выполнение инструкции приводит к переключению в режим ядра и запуску диспетчера системных сервисов. Номер системного сервиса передается в регистре процессора EAX, а регистр EDX указывает на список аргументов, предоставленных вызвавшим кодом. Для возврата в пользовательский режим диспетчер системных сервисов обычно выполняет инструкцию *sysexit*. (В некоторых случаях, например, когда в процессоре включен флаг *single-step*, диспетчер системных сервисов использует вместо *sysexit* инструкцию *iretd*.)

На 32-разрядных процессорах AMD K6 и выше Windows применяет специальную инструкцию *syscall*, которая функционирует аналогично x86-инструкции *sysenter*; Windows записывает в регистр процессора, связанный с инструкцией *syscall*, адрес диспетчера системных сервисов ядра. Номер системного вызова передается в регистре EAX, а в стеке хранятся аргументы, предоставленные вызвавшим кодом. После диспетчеризации ядро выполняет инструкцию *sysret*.

При загрузке Windows распознает тип процессора, на котором она работает, и выбирает подходящий системный код. Этот код для *NtReadFile* в пользовательском режиме выглядит так:

```
ntdll!NtReadFile:
77f5bfa8 b8b7000000      mov     eax,0xb7
77f5bfad ba0003fe7f      mov     edx,0x7ffe0300
77f5bfb2 ffd2            call    edx
77f5bfb4 c22400          ret     0x24
```

Номер системного сервиса — 0xb7 (183 в десятичной форме), инструкция вызова выполняет код диспетчеризации системного сервиса, установленный ядром, который в данном примере находится по адресу 0x7ffe0300. Поскольку пример взят для Pentium M, используется *sysenter*:

```
SharedUserData!SystemCallStub:
7ffe0300 8bd4            mov     edx,esp
7ffe0302 0f34            sysenter
7ffe0304 c3              ret
```

Диспетчеризация 64-разрядных системных сервисов

В архитектуре x64 операционная система Windows использует инструкцию *syscall*, которая работает аналогично инструкции *syscall* на процессорах AMD K6. Windows передает номер системного вызова в регистре EAX, первые четыре параметра в других регистрах, а остальные параметры (если они есть) в стеке:

```
ntdll!NtReadFile:
00000000'77f9fc60 4c8bd1          mov     r10,rcx
00000000'77f9fc63 b8bf000000      mov     eax,0xbf
00000000'77f9fc68 0f05            syscall
00000000'77f9fc6a c3              ret
```

В архитектуре IA64 для тех же целей применяется инструкция *epc* (Enter Privileged Mode). Первые восемь аргументов системного вызова передаются в регистрах, а остальное в стеке.

Диспетчеризация системных сервисов режима ядра

Как показано на рис. 3-14, ядро использует номер системного сервиса для поиска информации о нем в *таблице диспетчеризации системных сервисов* (system service dispatch table). Эта таблица похожа на описанную ранее таблицу IDT и отличается от нее тем, что каждый ее элемент содержит указатель на системный сервис, а не на процедуру обработки прерывания.

ПРИМЕЧАНИЕ Номера системных сервисов могут различаться в разных сервисных пакетах (service packs) — Microsoft время от времени добавляет или удаляет некоторые системные сервисы, а их номера генерируются автоматически при компиляции ядра.

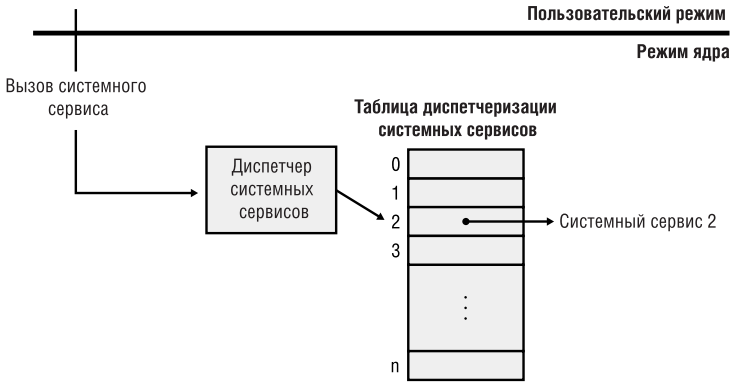


Рис. 3-14. *Исключения системных сервисов*

Диспетчер системных сервисов, *KiSystemService*, копирует аргументы вызвавшего кода из стека потока пользовательского режима в свой стек режима ядра (поэтому вызвавший код не может изменить значения аргументов после того, как они переданы ядру) и выполняет системный сервис. Если

переданные системному сервису аргументы содержат ссылки на буферы в пользовательском пространстве, код режима ядра проверяет возможность доступа к этим буферам, прежде чем копировать в них (или из них) данные.

Как будет показано в главе 6, у каждого потока есть указатель на таблицу системных сервисов. Windows располагает двумя встроенными таблицами системных сервисов, но поддерживает до четырех. Диспетчер системных сервисов определяет, в какой таблице содержится запрошенный сервис, интерпретируя 2-битное поле 32-битного номера системного сервиса как указатель на таблицу. Младшие 12 битов номера системного сервиса служат индексом внутри указанной таблицы. Эти поля показаны на рис. 3-15.

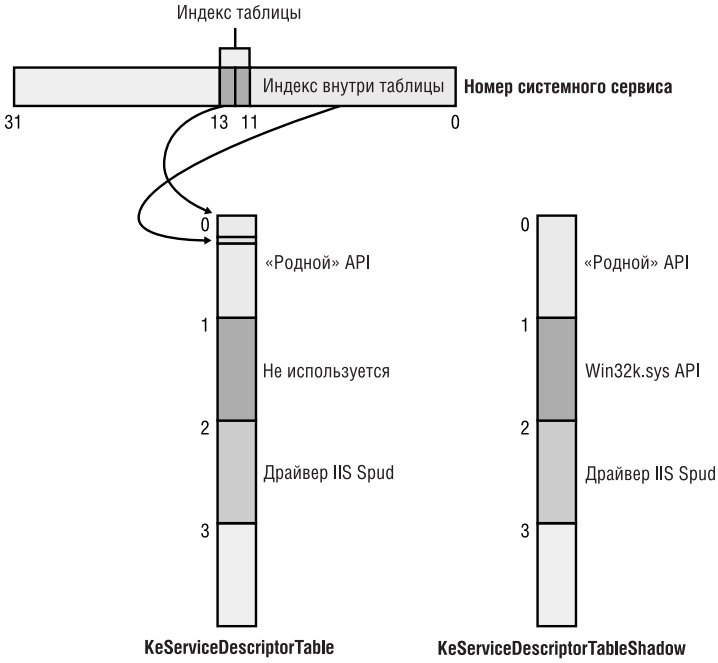


Рис. 3-15. *Трансляция номера системного сервиса*

Таблицы дескрипторов сервисов

Главная таблица по умолчанию, *KeServiceDescriptorTable*, определяет базовые сервисы исполнительной системы, реализованные в Ntoskrnl.exe. Другая таблица, *KeServiceDescriptorTableShadow*, включает в себя сервисы USER и GDI, реализованные в Win32k.sys — той части подсистемы Windows, которая работает в режиме ядра. Когда Windows-поток впервые вызывает сервис USER или GDI, адрес таблицы системных сервисов потока меняется на адрес таблицы, содержащей сервисы USER и GDI. Функция *KeAddSystemServiceTable* позволяет Win32k.sys и другим драйверам добавлять новые таблицы системных сервисов. Если в Windows 2000 установлены службы Internet Information Services (IIS), их драйвер поддержки (Spud.sys) после загрузки определяет

дополнительную таблицу сервисов. Так что после этого стороннее программное обеспечение может определить только одну дополнительную таблицу. Таблица сервисов, добавляемая *KeAddSystemServiceTable* (кроме таблицы Win32k.sys), копируется в таблицы *KeServiceDescriptorTable* и *KeServiceDescriptorTableShadow*. Windows поддерживает добавление лишь двух таблиц системных сервисов помимо главной и таблиц Win32.

ПРИМЕЧАНИЕ Windows Server 2003 Service Pack 1 и выше не поддерживает добавление таблиц системных сервисов, если не считать те, которые включаются Win32k.sys, так что этот способ не годится для расширения функциональности этой системы.

Инструкции для диспетчеризации сервисов исполнительной системы Windows содержатся в системной библиотеке Ntdll.dll. DLL-модули подсистем окружения вызывают функции из Ntdll.dll для реализации своих документированных функций. Исключением являются функции USER и GDI — здесь инструкции для диспетчеризации системных сервисов реализованы непосредственно в User32.dll и Gdi32.dll, а не в Ntdll.dll. Эти два случая иллюстрирует рис. 3-16.

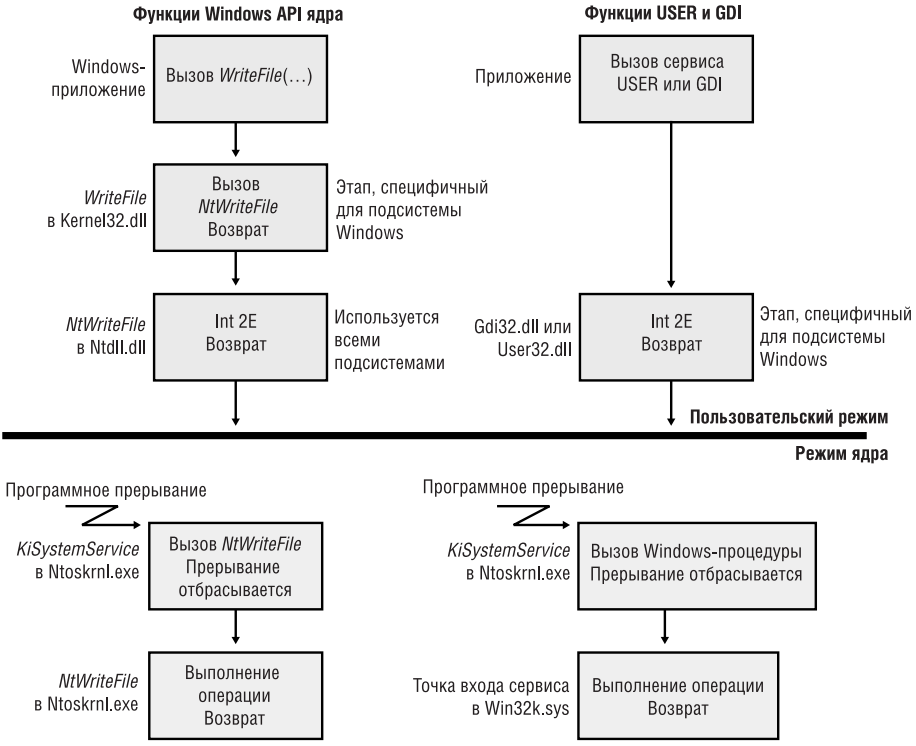


Рис. 3-16. *Диспетчеризация системных сервисов*

Как показано на рис. 3-16, Windows-функция *WriteFile* в Kernel32.dll вызывает функцию *NtWriteFile* из Ntdll.dll. Она в свою очередь выполняет соответствующую инструкцию, вызывающую срабатывание ловушки системного сервиса и передающую номер системного сервиса *NtWriteFile*. Далее диспетчер системных сервисов (функция *KiSystemService* в Ntoskrnl.exe) вызывает истинную *NtWriteFile* для обработки запроса на ввод-вывод. Для функций USER и GDI диспетчер системных сервисов вызывает функции из Win32k.sys, той части подсистемы Windows, которая работает в режиме ядра.

ЭКСПЕРИМЕНТ: наблюдение за частотой вызова системных сервисов

Вы можете наблюдать за частотой вызова системных сервисов с помощью счетчика System Calls/Sec (Системных вызовов/сек) объекта System (Система). Откройте оснастку Performance (Производительность) и щелкните кнопку Add (Добавить), чтобы добавить на график счетчик. Выберите объект System и счетчик System Calls/Sec, затем щелкните кнопки Add и Close (Закрыть).

Диспетчер объектов

Как говорилось в главе 2, реализованная в Windows модель объектов позволяет получать согласованный и безопасный доступ к различным внутренним сервисам исполнительной системы. В этом разделе описывается *диспетчер объектов* (object manager) — компонент исполнительной системы, отвечающий за создание, удаление, защиту и отслеживание объектов.

ЭКСПЕРИМЕНТ: исследование диспетчера объектов

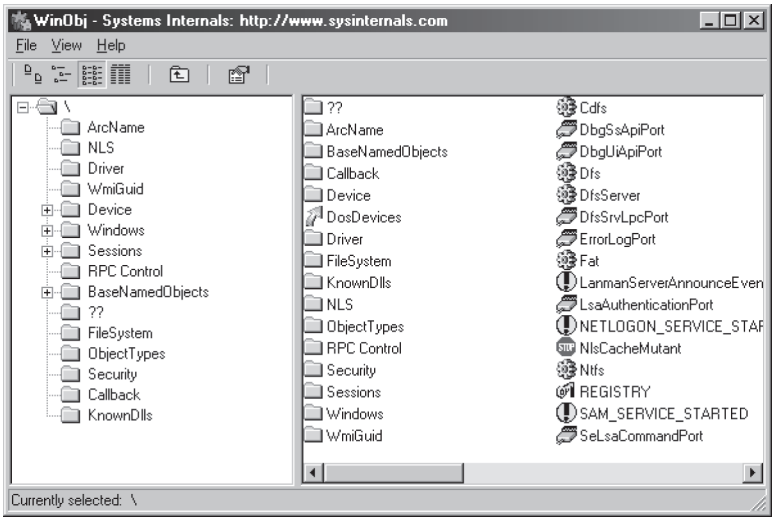
В этом разделе будут предлагаться эксперименты, которые покажут вам, как просмотреть базу данных диспетчера объектов. В них будут использоваться перечисленные ниже инструменты, которые вам нужно освоить (если вы их еще не освоили).

- Winobj можно скачать с сайта *www.sysinternals.com*. Она показывает пространство имен диспетчера объектов. Другая версия этой утилиты есть в Platform SDK (\Program Files\Microsoft Platform SDK\Bin\Winnt\Winobj.exe). Однако версия с *www.sysinternals.com* сообщает более детальную информацию об объектах (например, счетчик ссылок, число открытых описателей, дескрипторы защиты и т. д.).
- Process Explorer и Handle с сайта *www.sysinternals.com*. Отображают открытые описатели для процесса.
- Oh.exe (имеется в ресурсах Windows) выводит открытые описатели для процесса, но требует предварительной установки специального глобального флага.
- Команда *Openfiles /query* (в Windows XP и Windows Server 2003) отображает открытые описатели для процесса, но требует предварительной установки специального глобального флага.

см. след. стр.

- Команда *!handle* отладчика ядра отображает открытые описатели для процесса.

Средство просмотра объектов позволяет изучить пространство имен, поддерживаемое диспетчером объектов (имена есть не у всех объектов). Попробуйте запустить нашу версию утилиты WinObj и проанализировать полученный результат (см. иллюстрацию ниже).



Как уже отмечалось, утилита OH и команда *Openfiles /query* требуют установки глобального флага «поддержка списка объектов» (maintain objects list). (О глобальных флагах см. соответствующий раздел далее в этой главе.) OH установит этот флаг, если он еще не задан. Чтобы узнать, включен ли данный флаг, введите **Openfiles /Local**. Вы можете включить его командой *Openfiles /Local ON*. В любом случае нужно перезагрузить систему, чтобы новый параметр вступил в силу. Ни Process Explorer, ни Handle с сайта *www.sysinternals.com* не требуют включения слежения за объектами, потому что для получения соответствующей информации они используют драйвер устройства.

При разработке диспетчера объекта был выдвинут ряд требований, в соответствии с которыми он должен:

- реализовать общий, унифицированный механизм использования системных ресурсов;
- изолировать защиту объектов в одном участке операционной системы для соответствия требованиям безопасности класса C2;
- предоставлять механизм учета использования объектов процессами, позволяющий устанавливать лимиты на выделение процессам системных ресурсов;

- поддерживать такую схему именования объектов, которая позволяла бы легко включать как существующие объекты (устройства, файлы и каталоги файловой системы), так и независимые наборы объектов;
- соответствовать требованиям различных подсистем окружения операционной системы — например, поддерживать наследование ресурсов родительских процессов дочерними (необходимо в Windows и POSIX) и имена файлов, чувствительные к регистру букв (требуется в POSIX);
- устанавливать единообразные правила сохранения объектов в памяти (т. е. объект должен быть доступен, пока используется какими-либо процессами).

В Windows существует два вида внутренних объектов: *объекты исполнительной системы* (executive objects) и *объекты ядра* (kernel objects). Первые реализуются различными компонентами исполнительной системы (диспетчером процессов, диспетчером памяти, подсистемой ввода-вывода и т. д.). Вторые являются более примитивными объектами, которые реализуются ядром Windows. Эти объекты, невидимые коду пользовательского режима, создаются и используются только в исполнительной системе. Объекты ядра поддерживают фундаментальную функциональность (например, синхронизацию), на которую опираются объекты исполнительной системы. Так, многие объекты исполнительной системы содержат (инкапсулируют) один или несколько объектов ядра, как показано на рис. 3-17.

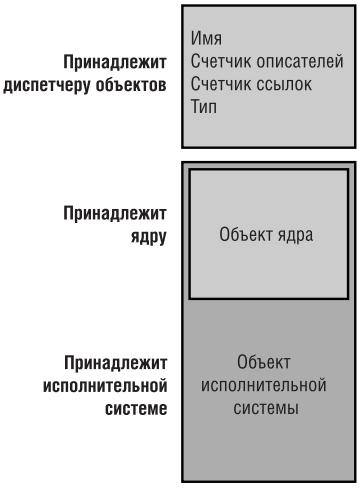


Рис. 3-17. *Объекты исполнительной системы, включающие объекты ядра*

Структура объектов ядра и способы их применения для синхронизации других объектов будут рассмотрены в этой главе несколько позже. А сейчас мы сосредоточимся на принципах работы диспетчера объектов, а также на структуре объектов исполнительной системы, описателях и таблицах описателей. Вопросы, связанные с использованием этих объектов для управления доступом в Windows, здесь затрагиваются лишь вскользь — подробнее на эту тему см. главу 8.

Объекты исполнительной системы

Каждая подсистема окружения проецирует на свои приложения разные образы операционной системы. Объекты исполнительной системы и сервисы объектов — именно те примитивы, из которых подсистемы окружения конструируют собственные версии объектов и других ресурсов.

Как правило, объекты исполнительной системы создаются подсистемой окружения в интересах пользовательских приложений или компонентов операционной системы в процессе обычной работы. Так, для создания файла Windows-приложение вызывает Windows-функцию *CreateFile*, реализованную в DLL подсистемы Windows, Kernel32.dll. После проверки и инициализации *CreateFile* в свою очередь вызывает *NtCreateFile*, встроенный сервис Windows, для создания объекта «файл» исполнительной системы.

Набор объектов, предоставляемый приложениям подсистемой окружения, может быть больше или меньше того набора, который предоставляется исполнительной системой. Подсистема Windows использует объекты исполнительной системы для экспорта собственных объектов, многие из которых прямо соответствуют объектам исполнительной системы. Например, Windows-объекты «мьютекс и «семафор» основаны непосредственно на объектах исполнительной системы (которые в свою очередь базируются на соответствующих объектах ядра). Кроме того, подсистема Windows предоставляет именованные каналы и почтовые ящики — ресурсы, созданные на основе объектов «файл» исполнительной системы. Некоторые подсистемы вроде POSIX вообще не поддерживают объекты как таковые. Подсистема POSIX использует объекты и сервисы исполнительной системы просто как основу для POSIX-процессов, каналов и других ресурсов, которые она предоставляет своим приложениям.

В таблице 3-3 кратко описываются основные объекты, предоставляемые исполнительной системой. Подробнее об объектах исполнительной системы см. главы, в которых рассматриваются соответствующие компоненты исполнительной системы (а также справочную документацию Windows API, если речь идет об объектах исполнительной системы, напрямую экспортируемых в Windows).

ПРИМЕЧАНИЕ В Windows 2000 исполнительная система реализует в общей сложности 27 типов объектов, а в Windows XP и Windows Server 2003 — 29. (В эти более новые версии Windows добавлены объекты DebugObject и KeyedEvent.) Многие из таких объектов предназначены только для использования компонентами исполнительной системы, которая и определяет их. Эти объекты недоступны Windows API напрямую. Пример таких объектов — Driver, Device и EventPair.

Таблица 3-3. *Объекты исполнительной системы, доступные Windows API*

Тип объектов	Представляет:
Символьная ссылка (symbolic link)	Механизм косвенной ссылки на имя объекта
Процесс (process)	Виртуальное адресное пространство и управляющую информацию, необходимую для выполнения набора объектов «поток»
Поток (thread)	Исполняемую часть процесса
Задание (job)	Совокупность процессов, управляемую как единая группа
Раздел (section)	Область разделяемой памяти (называемую объектом «проекция файла»)
Файл (file)	Экземпляр открытого файла или устройства ввода-вывода
Маркер доступа (access token)	Профиль защиты (SID, права пользователя и т. д.) процесса или потока
Событие (event)	Объект, который может пребывать либо в занятом, либо в свободном состоянии; используется для синхронизации или уведомления
Семафор (semaphore)	Счетчик, действующий как шлюз к ресурсам; позволяет указывать максимальное число потоков, которым разрешен доступ к защищенным этим объектом ресурсам
Мьютекс (mutex)	Механизм синхронизации, используемый для упорядочения доступа к ресурсам
Таймер (timer)	Механизм уведомления потока об истечении фиксированного периода времени
IoCompletion	Метод постановки в очередь и извлечения из нее уведомлений о завершении операций ввода-вывода (в Windows API называется портом завершения ввода-вывода)
Раздел реестра (key)	Механизм ссылки на данные в реестре; хотя разделы реестра появляются в пространстве имен диспетчера объектов, они управляются диспетчером конфигурации по аналогии с тем, как объекты «файл» управляются драйверами файловой системы; с объектом «раздел реестра» может быть сопоставлено произвольное количество параметров (от 0 и более)
WindowStation	Объект, содержащий буфер обмена, набор глобальных атомов и группу объектов «рабочий стол»
Рабочий стол (desktop)	Объект, содержащийся в объекте типа WindowStation; он описывает логическую поверхность экрана и содержит окна, меню и ловушки

ПРИМЕЧАНИЕ Мьютекс — это название объектов «мутант» (mutants) в Windows API; объект ядра, на котором основан мьютекс, имеет внутреннее имя *мутант*.

Структура объектов

Как показано на рис. 3-18, у каждого объекта есть заголовок и тело. Диспетчер объектов управляет заголовками объектов, а телами объектов управляют владеющие ими компоненты исполнительной системы. Кроме того, каждый заголовок объекта указывает на список процессов, которые открыли этот объект, и на специальный объект, называемый *объектом типа* (type object), — он содержит общую для всех экземпляров информацию.

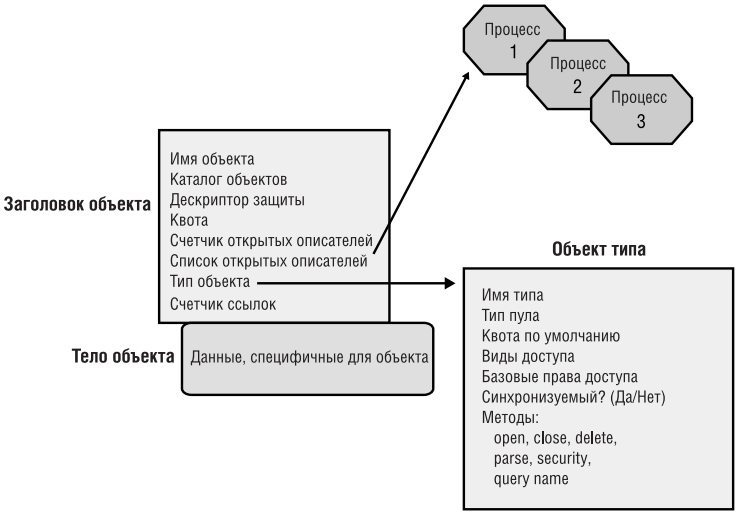


Рис. 3-18. *Структура объекта*

Заголовки и тела объектов

Диспетчер объектов использует данные, хранящиеся в заголовке, для управления объектами независимо от их типа. Стандартные атрибуты заголовка кратко описываются в таблице 3-4.

Таблица 3-4. *Стандартные атрибуты заголовка объекта*

Атрибут	Описание
Имя объекта (object name)	Позволяет совместно использовать объект, делая его видимым для других процессов
Каталог объектов (object directory)	Предоставляет иерархическую структуру для хранения имен объектов
Дескриптор защиты (security descriptor)	Определяет, кто и как может использовать данный объект (должен быть null для безымянных объектов)
Квота (quota charges)	Устанавливает ограничения на объемы ресурсов при открытии процессом описателя данного объекта
Счетчик открытых описателей (open handle count)	Подсчитывает, сколько раз был открыт описатель объекта

Таблица 3-4. (*окончание*)

Атрибут	Описание
Список открытых описателей (open handles list)	Указывает на список процессов, открывших описатели данного объекта
Тип объекта (object type)	Указывает на объект типа, содержащий атрибуты, общие для объектов данного типа
Счетчик ссылок (reference count)	Подсчитывает, сколько раз компоненты режима ядра ссылались на адрес данного объекта

Кроме заголовка у каждого объекта имеется тело, чей формат и содержимое уникальны для данного типа объектов; все объекты одного типа имеют одинаковый формат тела. Создавая тип объектов и предоставляя для него сервисы, компонент исполнительной системы может контролировать манипуляции с данными в телах всех объектов этого типа.

Диспетчер объектов предоставляет небольшой набор базовых сервисов, которые работают с атрибутами заголовка объекта и применимы к объектам любого типа (хотя некоторые базовые сервисы не имеют смысла для отдельных объектов). Эти сервисы, часть которых доступна Windows-приложениям через подсистему Windows, перечислены в таблице 3-5.

Базовые сервисы поддерживаются для всех типов объектов, но у каждого объекта есть свои сервисы для создания, открытия и запроса. Так, подсистема ввода-вывода реализует сервис создания файлов для объектов «файл», а диспетчер процессов — сервис создания процессов для объектов «процесс». Конечно, можно было бы реализовать единый сервис создания объектов, но подобная процедура оказалась бы весьма сложной, так как набор параметров, необходимых для инициализации объекта «файл», значительно отличается, скажем, от параметров инициализации объекта «процесс». А при вызове потоком сервиса объекта для определения его типа и обращении к соответствующей версии сервиса диспетчер объектов каждый раз сталкивался бы с необходимостью обработки дополнительных данных. В силу этих и иных причин сервисы, обеспечивающие создание, открытие и запросы, для каждого типа объектов реализованы отдельно.

Таблица 3-5. *Базовые сервисы объектов*

Сервис	Описание
Закрытие (close)	Закрывает описатель объекта
Дублирование (duplicate)	Позволяет совместно использовать объект за счет дублирования его описателя и передачи его другому процессу
Запрос объекта (query object)	Сообщает информацию о стандартных атрибутах объекта
Запрос защиты (query security)	Сообщает дескриптор защиты объекта
Установка защиты (set security)	Изменяет защиту объекта

см. след. стр.

Таблица 3-5. (*окончание*)

Сервис	Описание
Ожидание одного объекта (wait for a single object)	Синхронизирует выполнение потока с одним объектом
Ожидание нескольких объектов (wait for multiple objects)	Синхронизирует выполнение потока с несколькими объектами

Объекты типа

В заголовках объектов содержатся общие для всех объектов атрибуты, но их значения могут меняться у конкретных экземпляров данного объекта. Так, у каждого объекта есть уникальное имя и может быть уникальный дескриптор защиты. Однако некоторые атрибуты объектов остаются неизменными для всех объектов данного типа. Например, при открытии описателя объекта можно выбрать права доступа из набора, специфичного для объектов данного типа. Исполнительная система предоставляет в том числе атрибуты доступа «завершение» (terminate) и «приостановка» (suspend) для объектов «поток», а также «чтение» (read), «запись» (write), «дозапись» (append) и «удаление» (delete) для объектов «файл». Другой пример атрибута, специфичного для типа объектов, — синхронизация, о которой мы кратко расскажем ниже.

Чтобы сэкономить память, диспетчер объектов сохраняет статические атрибуты, специфичные для конкретного типа объектов, только при создании нового типа объектов. Для записи этих данных он использует собственный объект типа. Как показано на рис. 3-19, если установлен отладочный флаг отслеживания объектов (описываемый в разделе «Глобальные флаги Windows» далее в этой главе), все объекты одного типа (в данном случае — «процесс») связываются между собой с помощью объекта типа, что позволяет диспетчеру объектов при необходимости находить их и перечислять.

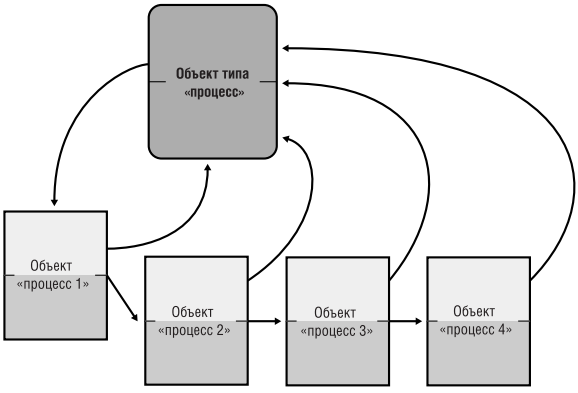
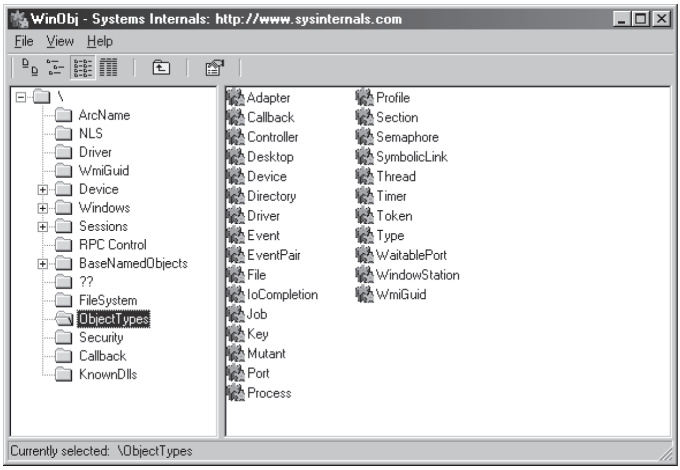


Рис. 3-19. *Объекты «процесс» и объект типа «процесс»*

ЭКСПЕРИМЕНТ: просмотр заголовков объектов и объектов типа

Вы можете увидеть список объектов типа, объявленных диспетчеру объектов, с помощью утилиты Winobj с сайта *www.sysinternals.com*. Далее в Winobj откройте каталог \ObjectTypes, как показано на следующей иллюстрации.



Чтобы просмотреть структуру данных типа объектов «процесс» в отладчике ядра, сначала идентифицируйте этот объект командой *!process*:

```
kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS 8a4ce668  SessionId: none  Cid: 0004
    Peb: 00000000  ParentCid: 0000
    DirBase: 00039000  ObjectTable: e1001c88  HandleCount: 474.
    Image: System
```

Затем выполните команду *!object*, указав адрес объекта «процесс» в качестве аргумента:

```
kd> !object 8a4ce668
Object: 8a4ce668  Type: (8a4ceca0) Process
    ObjectHeader: 8a4ce650
    HandleCount: 2  PointerCount: 89
```

Заметьте, что заголовок объекта начинается с 0x18 (24) байтов до нижней границы тела объекта. Увидеть заголовок объекта позволяет следующая команда:

```
kd> dt _object_header 8a4ce650
nt!_OBJECT_HEADER
   +0x000 PointerCount     : 79
   +0x004 HandleCount      : 2
   +0x004 NextToFree       : 0x00000002
```

см. след. стр.

```
   +0x008 Type             : 0x8a4ceca0
   +0x00c NameInfoOffset   : 0 ''
   +0x00d HandleInfoOffset : 0 ''
   +0x00e QuotaInfoOffset  : 0 ''
   +0x00f Flags            : 0x22 '"'
   +0x010 ObjectCreateInfo : 0x80545620
   +0x010 QuotaBlockCharged : 0x80545620
   +0x014 SecurityDescriptor : 0xe10001dc
   +0x018 Body             : _QUAD
```

Теперь посмотрите на структуру данных типа этих объектов, получив ее адрес из поля Type в структуре данных заголовка объекта:

```
kd> dt _object_type 8a4ceca0
ntdll!_OBJECT_TYPE
   +0x000 Mutex            : _ERESOURCE
   +0x038 TypeList         : _LIST_ENTRY
                             [ 0x8a4cecd8 - 0x8a4cecd8 ]
   +0x040 Name             : _UNICODE_STRING "Process"
   +0x048 DefaultObject    : (null)
   +0x04c Index            : 5
   +0x050 TotalNumberOfObjects : 0x30
   +0x054 TotalNumberOfHandles : 0x1b4
   +0x058 HighWaterNumberOfObjects : 0x3f
   +0x05c HighWaterNumberOfHandles : 0x1b8
   +0x060 TypeInfo         : _OBJECT_TYPE_INITIALIZER
   +0x0ac Key              : 0x636f7250
   +0x0b0 ObjectLocks      : [4] _ERESOURCE
```

Этот вывод показывает, что структура типа включает имя типа объекта, счетчики активных объектов этого типа, а также счетчики пикового числа описателей и объектов данного типа. В поле TypeInfo хранится указатель на структуру данных, в которой содержатся атрибуты, общие для всех объектов этого типа, а также указатели на методы типа:

```
kd> dt _object_type_initializer 8a4ceca0+60
ntdll!_OBJECT_TYPE_INITIALIZER
   +0x000 Length           : 0x4c
   +0x002 UseDefaultObject : 0 ''
   +0x003 CaseInsensitive  : 0 ''
   +0x004 InvalidAttributes : 0xb0
   +0x008 GenericMapping   : _GENERIC_MAPPING
   +0x018 ValidAccessMask  : 0x1f0fff
   +0x01c SecurityRequired : 0x1 ''
   +0x01d MaintainHandleCount : 0 ''
   +0x01e MaintainTypeList : 0 ''
   +0x020 PoolType         : 0 ( NonPagedPool )
   +0x024 DefaultPagedPoolCharge : 0x1000
   +0x028 DefaultNonPagedPoolCharge : 0x288
   +0x02c DumpProcedure    : (null)
```

```
+0x030 OpenProcedure    : (null)
+0x034 CloseProcedure   : (null)
+0x038 DeleteProcedure  : 0x805abe6e
  nt!PspProcessDelete+0
+0x03c ParseProcedure   : (null)
+0x040 SecurityProcedure : 0x805cf682
  nt!SeDefaultObjectMethod+0
+0x044 QueryNameProcedure : (null)
+0x048 OkayToCloseProcedure :  (null)
```

Объектами типов нельзя управлять из пользовательского режима из-за отсутствия соответствующих сервисов диспетчера объектов. Но некоторые из определяемых ими атрибутов видимы через отдельные системные сервисы и функции Windows API. Атрибуты, хранящиеся в объектах типа, описываются в таблице 3-6.

Таблица 3-6. *Атрибуты объекта типа*

Атрибут	Описание
Имя типа (type name)	Название объектов этого типа («процесс», «событие», «порт» и т. д.)
Тип пула (pool type)	Определяет тип памяти, выделяемый для объектов этого типа (подкачиваемая или неподкачиваемая)
Квота по умолчанию (default quota charges)	Ограничение по умолчанию на объемы подкачиваемой и неподкачиваемой памяти при открытии процессом описателя данного объекта
Виды доступа (access types)	Виды доступа, который поток может запрашивать при открытии описателя объекта этого типа («чтение», «запись», «завершение», «приостановка» и т. д.)
Базовые права доступа (generic access rights mapping)	Сопоставление четырех базовых прав доступа («чтение», «запись», «выполнение» и «все») с правами доступа, специфичными для данного типа
Синхронизация (synchronization)	Определяет, может ли поток ждать на объектах данного типа
Методы (methods)	Одна или несколько процедур, автоматически вызываемых диспетчером объектов на определенных этапах жизненного цикла объекта

Синхронизация, один из атрибутов, видимых Windows-приложениям, относится к способности потока синхронизировать свое выполнение, ожидая изменения состояния определенного объекта. Поток можно синхронизировать по таким объектам исполнительной системы, как задание, процесс, поток, файл, событие, семафор, мьютекс и таймер. Другие объекты исполнительной системы не поддерживают синхронизацию. Способность объекта к синхронизации зависит от того, содержит ли он встроенный объект диспетчера — объект ядра, который будет рассмотрен далее в этой главе.

Методы объекта

Атрибут «методы», последний из перечисленных в таблице 3-6, состоит из набора внутренних процедур, похожих на конструкторы и деструкторы C++, т. е. на процедуры, автоматически вызываемые при создании или уничтожении объекта. В диспетчере объектов эта идея получила дальнейшее развитие: он вызывает методы объекта и в других ситуациях, например при открытии или закрытии описателя объекта или при попытке изменения параметров защиты объекта. В некоторых типах объектов методы определяются в зависимости от того, как должен использоваться данный тип объектов.

При создании нового типа объектов компонент исполнительной системы может зарегистрировать у диспетчера объектов один или несколько методов. После этого диспетчер объектов вызывает методы на определенных этапах жизненного цикла объектов данного типа — обычно при их создании, удалении или модификации. Поддерживаемые диспетчером объектов методы перечислены в таблице 3-7.

Таблица 3-7. *Методы объекта*

Метод	Когда происходит вызов метода
Open	При открытии описателя объекта
Close	При закрытии описателя объекта
Delete	Перед удалением объекта диспетчером объектов
Query name	При запросе потоком имени объекта (например, файла), существующего во вторичном пространстве имен объектов
Parse	При поиске диспетчером объектов имени, существующего во вторичном пространстве имен объектов
Security	При считывании или изменении процессом параметров защиты объекта, например файла, существующего во вторичном пространстве имен объектов

Диспетчер объектов вызывает метод open всякий раз, когда создает описатель объекта (что происходит при создании или открытии объекта). Однако метод open определен только в одном типе объектов — WindowStation. Этот метод необходим таким объектам для того, чтобы Win32k.sys мог использовать часть памяти совместно с процессом, который служит в качестве пула памяти, связанного с объектом «рабочий стол».

Пример использования метода close можно найти в подсистеме ввода-вывода. Диспетчер ввода-вывода регистрирует метод close для объектов типа «файл», а диспетчер объектов вызывает метод close при закрытии каждого описателя объекта этого типа. Метод close проверяет, не осталось ли у процесса, закрывающего описатель файла, каких-либо блокировок для файла, и, если таковые есть, снимает их. Диспетчер объектов не может и не должен самостоятельно проверять наличие блокировок для файла.

Перед удалением временного объекта из памяти диспетчер объектов вызывает метод delete, если он зарегистрирован. Например, диспетчер памяти регистрирует для объектов типа «раздел» метод delete, освобождающий фи-

зические страницы, используемые данным разделом. Перед удалением объекта «раздел» этот метод также проверяет различные внутренние структуры данных, выделенные для раздела диспетчером памяти. Диспетчер объектов не мог бы сделать эту работу, поскольку он ничего не знает о внутреннем устройстве диспетчера памяти. Методы delete других объектов выполняют аналогичные функции.

Если диспетчер объектов находит существующий вне его пространства имен объект, метод parse (по аналогии с методом query name) позволяет этому диспетчеру передавать управление вторичному диспетчеру объектов. Когда диспетчер объектов анализирует имя объекта, он приостанавливает анализ, встретив объект с сопоставленным методом parse, и вызывает метод parse, передавая ему оставшуюся часть строки с именем объекта. Кроме пространства имен диспетчера объектов, в Windows есть еще два пространства имен: реестра (реализуемое диспетчером конфигурации) и файловой системы (реализуемое диспетчером ввода-вывода через драйверы файловой системы). О диспетчере конфигурации см. главу 5; о диспетчере ввода-вывода и драйверах файловой системы см. главу 9.

Например, когда процесс открывает описатель объекта с именем \Device\Floppy0\docs\resume.doc, диспетчер объектов просматривает свое дерево имен и ищет объект с именем Floppy0. Обнаружив, что с этим объектом сопоставлен метод parse, он вызывает его, передавая ему остаток строки с именем объекта (в данном случае — строку \docs\resume.doc). Метод parse объектов «устройство» (device objects) является процедурой ввода-вывода, которая регистрируется диспетчером ввода-вывода при определении типа объекта «устройство». Процедура parse диспетчера ввода-вывода принимает строку с именем и передает ее соответствующей файловой системе, которая ищет файл на диске и открывает его.

Подсистема ввода-вывода также использует метод security, аналогичный методу parse. Он вызывается каждый раз, когда поток пытается запросить или изменить параметры защиты файла. Эта информация для файлов отличается от таковой для других объектов, поскольку хранится в самом файле, а не в памяти. Поэтому для поиска, считывания или изменения параметров защиты нужно обращаться к подсистеме ввода-вывода.

Описатели объектов и таблица описателей, принадлежащая процессу

Когда процесс создает или открывает объект по имени, он получает *описатель* (handle), который дает ему доступ к объекту. Ссылка на объект по описателю работает быстрее, чем по имени, так как при этом диспетчер объектов может сразу найти объект, не просматривая список имен. Процессы также могут получать описатели объектов, во-первых, путем их наследования в момент своего создания (если процесс-создатель разрешает это, указывая соответствующий флаг при вызове *CreateProcess*, и если описатель помечен как наследуемый либо в момент создания, либо позднее, вызовом Windows-функции *SetHandleInformation*), а во-вторых, за счет приема дублированного описателя от другого процесса (см. описание Windows-функции *DuplicateHandle*).

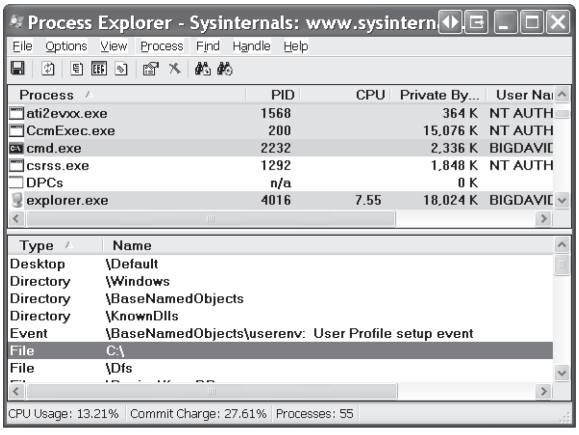
Чтобы потоки процесса пользовательского режима могли оперировать объектом, им нужен описатель этого объекта. Идея применения описателей для управления ресурсами сама по себе не нова. Например, стандартные библиотеки языков C, Pascal (и других) при открытии файла возвращают его описатель. Описатели служат косвенными указателями на системные ресурсы, что позволяет прикладным программам избегать прямого взаимодействия с системными структурами данных.

ПРИМЕЧАНИЕ Компоненты исполнительной системы и драйверы устройств могут обращаться к объектам напрямую, поскольку выполняются в режиме ядра и ввиду этого имеют доступ к структурам объектов в системной памяти. Однако они должны объявлять о своем использовании объектов, увеличивая значение счетчика ссылок, что гарантирует сохранность нужного объекта (см. раздел «Хранение объектов в памяти» далее в этой главе).

Описатели дают и другие преимущества. Во-первых, описатели файлов, событий или процессов совершенно одинаковы — просто ссылаются на разные объекты. Это дает возможность создать унифицированный интерфейс для ссылок на объекты любого типа. Во-вторых, только диспетчер объектов имеет право физически создавать описатели и искать их объекты, а значит, он может проанализировать любое действие с объектом в пользовательском режиме и решить, позволяет ли профиль защиты вызывающей программы выполнить над объектом запрошенную операцию.

ЭКСПЕРИМЕНТ: просмотр открытых описателей

Запустите Process Explorer и убедитесь, что в его окне нижняя секция включена и настроена на отображение открытых описателей. (Выберите View, Lower Pane View и Handles.) Затем откройте окно командной строки и просмотрите таблицу описателей для нового процесса Cmd.exe. Вы должны увидеть открытый описатель файла — текущего каталога. Например, если текущий каталог — C:\, Process Explorer выводит следующее.



Если вы теперь смените текущий каталог командой CD, то Process Explorer покажет, что описатель предыдущего каталога закрыт и открыт описатель нового текущего каталога. Предыдущий описатель ненадолго выделяется красным цветом, а новый — зеленым. Длительность подсветки настраивается щелчком кнопки Options и регулировкой параметра Difference Highlight Duration.

Такая функциональность Process Explorer упрощает наблюдение за изменениями в таблице описателей. Например, если в процессе происходит утечка описателей, просмотр таблицы описателей в Process Explorer позволяет быстро увидеть, какой описатель (или описатели) открывается, но не закрывается. Эта информация помогает программисту обнаружить утечку описателей.

Таблицу открытых описателей также можно вывести, используя командную строку утилиты Handle (*www.sysinternals.com*). Например, обратите внимание на следующий фрагмент вывода Handle, где показана таблица описателей для процесса Cmd.exe до и после смены каталога:

```
C:\>handle -p cmd.exe
Handle v2.2
Copyright (C) 1997-2004 Mark Russinovich
Sysinternals - www.sysinternals.com
--------------------------------
cmd.exe pid: 3184 BIGDAVID\dsolomon
   b0: File          C:\
C:\>cd windows
C:\WINDOWS>handle -p cmd.exe
.
.
cmd.exe pid: 3184 BIGDAVID\dsolomon
   b4: File          C:\WINDOWS
```

Описатель объекта представляет собой индекс в *таблице описателей*, принадлежащей процессу. На нее указывает блок процесса исполнительной системы (EPROCESS), рассматриваемый в главе 6. Индекс первого описателя равен 4, второго — 8 и т. д. Таблица содержит указатели на все объекты, описатели которых открыты данным процессом. Эти таблицы реализованы по трехуровневой схеме аналогично тому, как блок управления памятью в системах типа x86 реализует трансляцию виртуальных адресов в физические, поддерживая более 16 000 000 описателей на каждый процесс (см. главу 7).

При создании процесса в Windows 2000 диспетчер объектов формирует верхний уровень таблицы описателей, содержащий указатели на таблицы среднего уровня; средний уровень содержит первый массив указателей на таблицы вторичных описателей, а нижний — первую таблицу вторичных описателей. На рис. 3-20 показана структура таблицы описателей в Windows 2000. В этой операционной системе диспетчер объектов интерпретирует младшие 24 бита описателя объекта как три 8-битных поля, являющиеся

индексами для каждого из трех уровней таблицы описателей. В Windows XP и Windows Server 2003 при создании процесса создается лишь таблица описателей нижнего уровня — остальные уровни формируются по мере необходимости. В Windows 2000 таблица вторичных описателей состоит из 255 элементов. В Windows XP и Windows Server 2003 такая же таблица включает столько элементов, сколько помещается на страницу памяти минус один элемент, который используется для аудита описателей (handle auditing). Например, на x86-системах размер страницы составляет 4096 байтов. Делим это значение на размер элемента (8 байтов), вычитаем 1 и получаем всего 511 элементов в таблице описателей нижнего уровня. Наконец, таблица описателей среднего уровня в Windows XP и Windows Server 2003 содержит полную страницу указателей на таблицы вторичных описателей, поэтому количество таблиц вторичных описателей зависит от размеров страницы и указателя на конкретной аппаратной платформе.

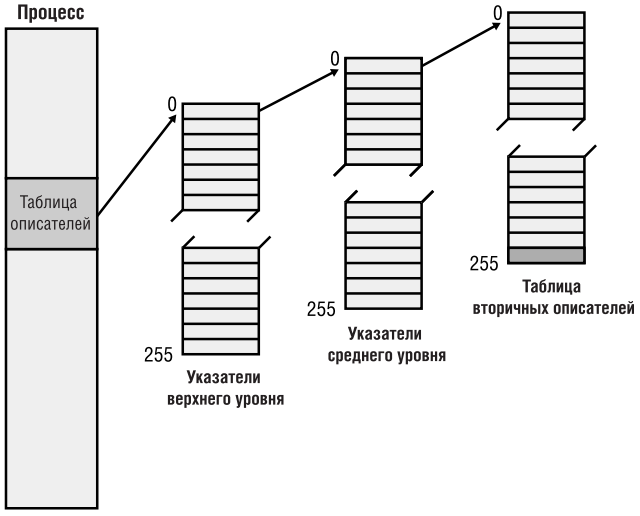


Рис. 3-20. *Структура таблицы описателей, принадлежащая процессу в Windows 2000*

ЭКСПЕРИМЕНТ: создание максимального количества описателей

Тестовая программа Testlimit (*wwwsysinternals.com/windowsinternals.shtml*) позволяет открывать описатели объекта до тех пор, пока не будет исчерпан их лимит. С ее помощью вы увидите, сколько описателей можно создать в одном процессе в вашей системе. Поскольку память под таблицу описателей выделяется из пула подкачиваемых страниц, этот пул может быть исчерпан до того, как вы достигнете предельного числа описателей, допустимых в одном процессе.

1. Скачайте файл Testlimit.zip по только что упомянутой ссылке и разархивируйте его в какой-нибудь каталог.

2. Запустите Process Explorer, щелкните View, затем System Information. Обратите внимание на текущий и максимальный размеры пула подкачиваемой памяти. (Для вывода максимальных размеров пулов Process Explorer нужно настроить на доступ к символам для образа ядра, Ntoskrnl.exe.) Пусть эта информация отображается, когда вы запустите программу Testlimit.
3. Откройте окно командной строки.
4. Запустите программу Testlimit с ключом *–h* (для этого введите **testlimit -h**). Когда Testlimit не удастся открыть очередной описатель, она сообщит общее число созданных ею описателей. Если это значение окажется меньше, чем около 16 миллионов, то, вероятно, вы исчерпали пул подкачиваемой памяти до достижения теоретически предельного числа описателей.
5. Закройте окно командной строки; это приведет к завершению процесса Testlimit и автоматическому закрытию всех открытых описателей.

Как показано на рис. 3-21, в x86-системах каждый элемент таблицы описателей состоит из структуры с двумя 32-битными элементами: указателем на объект (с флагами) и маской доступа. В 64-разрядных системах каждый элемент имеет размер 12 байтов и состоит из 64-битного указателя на заголовок объекта и 32-битной маски доступа (маски доступа описываются в главе 8).

В Windows 2000 первый 32-битный элемент содержит указатель на заголовок объекта и четыре флага. Поскольку заголовки объектов всегда выравниваются по границе 8 байтов, в качестве флагов используются младшие 3 бита и один старший. Старший бит является флагом блокировки (lock). Когда диспетчер объектов транслирует описатель в указатель на объект, он блокирует соответствующую запись на время трансляции. Так как все объекты находятся в системном адресном пространстве, старший бит указателя на объект устанавливается в 1. (Гарантируется, что адреса всегда будут превышать 0x80000000 даже на системах, запускаемых с ключом загрузки /3GB.) Так что, пока элемент таблицы описателей не заблокирован, диспетчер объектов может сбросить старший бит в 0. При блокировке элемента таблицы диспетчер объектов устанавливает этот бит и получает корректное значение указателя. Диспетчер блокирует всю таблицу описателей (используя специальную блокировку, сопоставляемую с каждым процессом), только если процесс создает новый описатель или закрывает существующий. В Windows XP и Windows Server 2003 флаг блокировки хранится в младшем бите указателя на объект. А флаг, который в Windows 2000 хранился в этом бите, теперь хранится в ранее зарезервированном бите маски доступа.

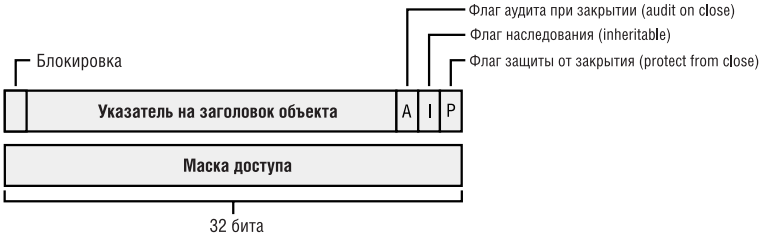


Рис. 3-21. *Структура элемента таблицы описателей*

Первый флаг указывает, имеет ли право вызывающая программа закрывать данный описатель. Второй определяет, получат ли процессы, созданные данным процессом, копию этого описателя. (Как уже отмечалось, наследование описателя можно указать при его создании или позже, через Windows-функцию *SetHandleInformation*. Этот флаг тоже можно задать вызовом *SetHandleInformation*.) Третий задает, будет ли генерироваться сообщение аудита при закрытии объекта. (Этот флаг не предоставляется в Windows и предназначен для внутреннего использования диспетчером объектов.)

Системным компонентам и драйверам устройств зачастую нужно открывать описатели объектов, доступа к которым у приложений пользовательского режима нет. Это достигается созданием описателей в *таблице описателей ядра* (kernel handle table) (внутреннее имя — *ObpKernelHandleTable*). Описатели в этой таблице доступны только в режиме ядра в контексте любого процесса. Это значит, что функции режима ядра могут ссылаться на эти описатели из контекста любого процесса без ущерба для производительности. Диспетчер объектов распознает ссылки на описатели в таблице описателей ядра, когда старший бит в них установлен, т. е. когда в таких ссылках содержатся значения, превышающие 0x80000000. В Windows 2000 таблица описателей ядра является независимой таблицей описателей, но в Windows XP и Windows Server 2003 она служит и таблицей описателей для процесса System.

ЭКСПЕРИМЕНТ: просмотр таблицы описателей с помощью отладчика ядра

Команда *!handle* отладчика ядра допускает следующие аргументы:

```
!handle <индекс_описателя> <флаги> <идентификатор_процесса>
```

Индекс описателя определяет элемент в таблице описателей (0 — вывод всех описателей). Индекс первого описателя равен 4, второго — 8 и т. д. Например, введя *!handle 4*, вы увидите первый описатель в текущем процессе.

Вы можете указывать флаги, являющиеся битовыми масками, где бит 0 означает, что нужно вывести лишь информацию из элемента таблицы, бит 1 — показать не только используемые, но и свободные описатели, а бит 2 — сообщить информацию об объекте, на который ссылается описатель. Следующая команда выводит полную информацию о таблице описателей в процессе с идентификатором 0x408.

```
kd> !handle 0 7 408

processor number 0
Searching for Process with Cid == 408
PROCESS 865f0790 SessionId: 0 Cid: 0408 Peb: 7ffdf000
    ParentCid: 01dc DirBase: 04fd3000  ObjectTable: 856ca888
    TableSize:  21.
    Image: i386kd.exe

Handle Table at e2125000 with 21 Entries in use
0000: free handle
0004: Object: e20da2e0  GrantedAccess: 000f001f
Object: e20da2e0  Type: (81491b80) Section
    ObjectHeader: e20da2c8
        HandleCount: 1  PointerCount: 1

0008: Object: 80b13330  GrantedAccess: 00100003
Object: 80b13330  Type: (81495100) Event
    ObjectHeader: 80b13318
        HandleCount: 1  PointerCount: 1
```

Защита объектов

Открывая файл, нужно указать, для чего это делается — для чтения или записи. Если вы попытаетесь записать что-нибудь в файл, открытый для чтения, то получите ошибку. Аналогичным образом действует и исполнительная система: когда процесс создает объект или открывает описатель существующего объекта, он должен указывать набор *желательных прав доступа* (desired access rights), сообщая тем самым, что именно он собирается делать с объектом. Процесс может запросить либо набор стандартных прав доступа (чтение, запись, выполнение), применимых ко всем объектам, либо специфические права доступа, различные для объектов разного типа. Так, в случае объекта «файл» процесс может запросить права на удаление файла или дозапись, а в случае объекта «поток» — права на приостановку потока или его завершение.

Когда процесс открывает описатель объекта, диспетчер объектов вызывает так называемый *монитор состояния защиты*[*] (security reference monitor), часть подсистемы защиты, работающую в режиме ядра, и посылает ему уведомление о наборе желательных для процесса прав доступа. Монитор состояния защиты проверяет, разрешает ли дескриптор защиты объекта запрашиваемый тип доступа. Если да, монитор состояния защиты возвращает процессу набор *предоставленных прав доступа* (granted access rights), информацию о которых диспетчер объектов сохраняет в создаваемом им описателе объекта. Как подсистема защиты определяет, кто и к каким объектам может получать доступ, рассматривается в главе 8.

[*] На самом деле этот компонент представляет собой нечто вроде монитора запросов к подсистеме защиты. — *Прим. перев.*

После этого всякий раз, когда потоки процесса используют описатель, диспетчер объектов может быстро проверить, соответствует ли набор предоставленных прав доступа, хранящихся в описателе, действиям, которые намеревается выполнить вызванный потоками сервис объекта. Так, если вызывающая программа запросила доступ для чтения к объекту «раздел», а затем вызывает сервис для записи в этот объект, последний сервис не выполняется.

Хранение объектов в памяти

Объекты бывают двух типов: временные (temporary) и постоянные (permanent). Большинство объектов временные, т. е. они хранятся, пока используются, и освобождаются, как только необходимость в них отпадает. Постоянные объекты существуют до тех пор, пока они не освобождаются явным образом. Поскольку большинство объектов временные, остальная часть этого раздела будет посвящена тому, как диспетчер объектов реализует *хранение объектов в памяти* (object retention), т. е. сохранение временных объектов лишь до тех пор, пока они используются, с их последующим удалением. Так как для доступа к объекту все процессы пользовательского режима должны сначала открыть его описатель, диспетчер объектов может легко отслеживать, сколько процессов и даже какие именно из них используют объект. Учет описателей является одним из механизмов, реализующих хранение объектов в памяти. Этот механизм двухфазный. Первая фаза называется *хранением имен* (name retention) и контролируется числом открытых описателей объекта. Каждый раз, когда процесс открывает описатель объекта, диспетчер увеличивает значение счетчика открытых описателей в заголовке объекта. По мере того как процессы завершают использование объекта и закрывают его описатели, диспетчер уменьшает значение этого счетчика. Когда счетчик обнуляется, диспетчер удаляет имя объекта из своего глобального пространства имен. После этого новые процессы уже не смогут открывать описатели данного объекта.

Вторая фаза заключается в том, что прекращается хранение тех объектов, которые больше не используются (т. е. они удаляются). Так как код операционной системы обычно обращается к объектам по указателям, а не описателям, диспетчер объектов должен регистрировать и число указателей объектов, переданных процессам операционной системы. При каждой выдаче указателя на объект он увеличивает значение *счетчика ссылок* на объект. Компоненты режима ядра, прекратив использовать указатель, вызывают диспетчер объектов для уменьшения счетчика ссылок. Система также увеличивает счетчик ссылок при увеличении счетчика описателей, а при уменьшении счетчика описателей соответственно уменьшает счетчик ссылок, поскольку описатель тоже является подлежащей учету ссылкой на объект (подробнее об этих механизмах см. описание функции *ObReferenceObjectByPointer* или *ObDereferenceObject* в DDK).

На рис. 3-22 показаны два задействованных объекта-события. Процесс A открыл первый объект, а процесс B — оба объекта. Кроме того, на первый объект ссылается какая-то структура режима ядра, и его счетчик ссылок ра-

вен 3. Так что, даже если оба процесса закроют свои описатели первого объекта, он по-прежнему будет существовать, поскольку его счетчик ссылок еще не обнулится. Но, когда процесс B закроет свой описатель второго объекта, этот объект будет удален.

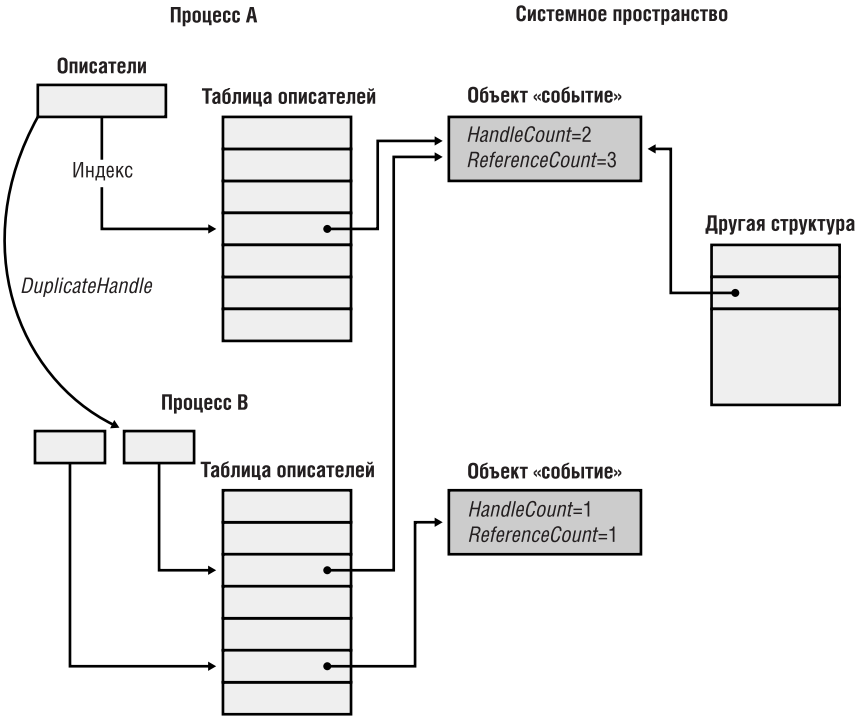


Рис. 3-22. *Счетчики описателей и ссылок*

Таким образом, даже если счетчик открытых описателей объекта обнулится, счетчик ссылок может превышать нулевое значение, указывая, что операционная система еще использует объект. В конце концов счетчик ссылок тоже обнулится, и тогда диспетчер удалит соответствующий объект из памяти.

Такой механизм позволяет хранить объект и его имя в памяти, просто не закрывая его описатель. Программистам, создающим приложения с двумя и более взаимодействующими процессами, не приходится беспокоиться о том, что один из процессов удалит объект в то время, когда он еще используется другим процессом. Кроме того, закрытие описателей объекта, принадлежащих приложению, еще не означает, что этот объект будет немедленно удален, — он может использоваться операционной системой. Например, какой-то процесс создает второй процесс для выполнения программы в фоновом режиме, после чего немедленно закрывает описатель созданного процесса. Так как второй процесс еще не закончил свою работу и выполняется операционной системой, она поддерживает ссылку на этот объект «процесс». Дис-

петчер сможет обнулить счетчик ссылок на второй процесс и удалить его, только когда завершится выполнение этого процесса.

Учет ресурсов

Учет ресурсов, как и хранение объектов, тесно связан с использованием описателей объектов. Положительное значение счетчика открытых описателей указывает на то, что данный ресурс задействован какими-то процессами. Когда счетчик описателей и счетчик ссылок на некий объект обнуляются, процессы, использовавшие этот объект, больше не занимают память, отведенную под него.

Во многих операционных системах для ограничения доступа процессов к системным ресурсам применяется система квот. Однако типы устанавливаемых для процессов квот иногда весьма разнообразны, а отслеживающий квоты код распределен по всей операционной системе. Так, в некоторых операционных системах компонент ввода-вывода может регистрировать и ограничивать число файлов, которые может открыть процесс, а компонент управления памятью может накладывать ограничения на объем памяти, выделяемой потокам процесса. Компонент, отвечающий за управление процессами, способен ограничивать максимальное число новых процессов или новых потоков процесса. Каждое из этих ограничений отслеживается и реализуется в различных частях операционной системы.

Диспетчер объектов Windows, напротив, представляет собой компонент централизованного учета ресурсов. В заголовке каждого объекта содержится атрибут квоты, определяющий, насколько диспетчер объектов уменьшает квоту подкачиваемой или неподкачиваемой памяти процесса при открытии его потоком описателя этого объекта.

У каждого процесса в Windows имеется структура квот, регистрирующая лимиты и текущее количество используемой памяти из подкачиваемого и неподкачиваемого пулов, а также из страничного файла. (Введите **dt nt!_EPROCESS_QUOTA_ENTRY** в отладчике ядра, чтобы увидеть формат этой структуры.) Значения данных квот по умолчанию равны 0 (ограничений нет), но их можно указать, модифицировав параметры в реестре (см. параметры NonPagedPoolQuota, PagedPoolQuota и PagingFileQuota в разделе HKLM\System\CurrentControlSet\Session Manager\Memory Management). Заметьте, что все процессы в интерактивном сеансе используют один и тот же блок квот (документированного способа создания процессов с собственными блоками квот нет).

Имена объектов

Важное условие для создания множества объектов — эффективная система учета. Для учета диспетчеру объектов нужна следующая информация:

- способ, которым можно было бы отличать один объект от другого;
- метод поиска и получения конкретного объекта.

Первое требование реализуется за счет присвоения имен объектам. Это расширение обычной для большинства операционных систем функциональности, в которых отдельным системным ресурсам, например файлам, каналам или блокам разделяемой памяти, можно присваивать имена. Исполнительная система, напротив, позволяет именовать любой объект, представляющий ресурс. Второе требование (поиск и получение объектов) также реализуется через именование объектов. Если диспетчер хранит объекты в соответствии с их именами, он может быстро найти объект по его имени.

Имена объектов отвечают и третьему требованию, не упомянутому в предыдущем списке: процессам должна быть предоставлена возможность совместного использования объектов. Пространство имен объектов исполнительной системы является глобальным, видимым любому процессу в системе. Если один процесс создает объект и помещает его имя в глобальное пространство имен, то другой процесс может открыть описатель этого объекта, указав нужное имя. Если объект не предназначен для совместного использования, процесс-создатель просто не присваивает ему имя.

Для большей эффективности диспетчер объектов не ищет имя объекта при каждой попытке его использования. Поиск по имени ведется только в двух случаях. Во-первых, при создании процессом именованного объекта: перед тем как сохранить имя объекта в глобальном пространстве имен, диспетчер проверяет, нет ли в нем такого же имени. Во-вторых, открывая описатель именованного объекта, диспетчер ищет объект по имени и возвращает его описатель, который затем используется для ссылки на объект. Диспетчер позволяет выбирать, надо ли при поиске учитывать регистр букв. Эта функциональность поддерживается POSIX и другими подсистемами окружения, в которых имена файлов чувствительны к регистру букв.

Где именно хранятся имена объектов, зависит от типа объектов. В таблице 3-8 перечислены стандартные каталоги объектов, имеющиеся на всех системах под управлением Windows. Пользовательским программам видны только каталоги \BaseNamedObjects и \GLOBAL?? (\?? в Windows 2000).

Поскольку имена базовых объектов ядра вроде мьютексов, событий, семафоров, ожидаемых таймеров и разделов хранятся в одном каталоге, они не должны совпадать, даже если относятся к объектам разных типов. Это ограничение подчеркивает, насколько осторожно надо выбирать имена, чтобы они не конфликтовали с другими (используйте, например, префиксы имен в виде названия вашей компании и программного продукта).

Таблица 3-8. *Стандартные каталоги объектов*

Каталог	Типы объектов, имена которых хранятся в каталоге
\GLOBAL?? (\?? в Windows 2000)	Имена устройств MS-DOS (\DosDevices является символьной ссылкой на этот каталог)
\BaseNamedObjects	Мьютексы, события, семафоры, ожидаемые таймеры и разделы
\Callback	Объекты обратного вызова
\Device	Объекты устройств

см. след. стр.

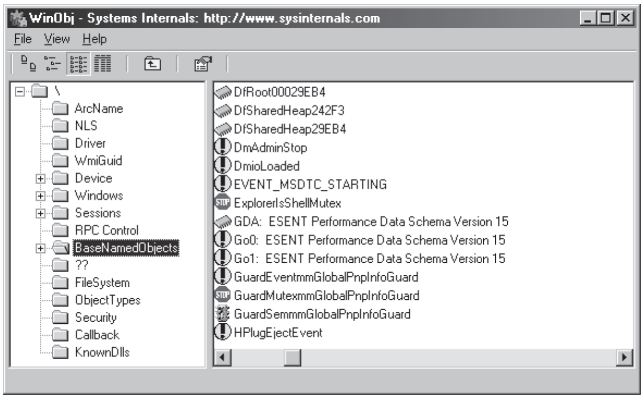
Таблица 3-8. (*окончание*)

Каталог	Типы объектов, имена которых хранятся в каталоге
\Driver	Объекты драйверов
\FileSystem	Объекты драйверов файловых систем и объекты распознавания файловых систем
\KnownDlls	Имена разделов и пути поиска известных DLL (DLL, проецируемых системой при запуске)
\Nls	Имена разделов спроецированных таблиц поддержки национальных языков
\ObjectTypes	Имена типов объектов
\RPC Control	Объекты портов, используемые для вызова удаленных процедур (RPC)
\Security	Имена объектов, специфичных для системы защиты
\Windows	Порты подсистемы Windows и объекты WindowStation

Имена объектов глобальны в пределах компьютера (или всех процессоров на многопроцессорной системе) и невидимы через сеть. Однако метод parse диспетчера объектов позволяет получать доступ к именованным объектам, существующим на других компьютерах. Так, диспетчер ввода-вывода, предоставляющий сервисы объектов «файл», расширяет функции диспетчера объектов для работы с файлами на удаленных компьютерах. При запросе на открытие объекта «файл» на удаленном компьютере диспетчер объектов вызывает метод parse, что позволяет диспетчеру ввода-вывода перехватить запрос и направить его сетевому редиректору — драйверу, обращающемуся к файлам через сеть. Серверный код на удаленной Windows-системе вызывает диспетчер объектов и диспетчер ввода-вывода на этой системе для поиска нужного объекта «файл» и возврата данных через сеть.

ЭКСПЕРИМЕНТ: просмотр именованных базовых объектов

Список именованных базовых объектов можно просмотреть с помощью утилиты Winobj (*www.sysinternals.com*). Запустите Winobj.exe и щелкните каталог \BaseNamedObjects, как показано ниже.



Именованные объекты отображаются справа. Тип объектов обозначается следующими значками:

- «stop» — мьютексы;
- в виде микросхем памяти — разделы (объекты «проекция файла»);
- в виде восклицательного знака — события;
- похожие на светофоры — семафоры;
- в виде изогнутой стрелки — символьные ссылки.

Объекты «каталоги объектов» (object directory objects) С помощью этих объектов диспетчер объектов поддерживает иерархическую структуру пространства имен. Этот объект аналогичен каталогу файловой системы и содержит имена других объектов, а также другие каталоги объектов. Он включает информацию, достаточную для трансляции имен объектов в указатели на сами объекты. Диспетчер использует указатели для создания описателей объектов, возвращаемых программам пользовательского режима. Каталоги для хранения объектов могут создаваться как кодом режима ядра (включая компоненты исполнительной системы и драйверы устройств), так и кодом пользовательского режима (в том числе подсистемами). Например, диспетчер ввода-вывода создает каталог объектов \Device с именами объектов, представляющих устройства ввода-вывода.

Символьные ссылки (symbolic links) В некоторых файловых системах (например, NTFS и отдельных UNIX-системах) с помощью символьной ссылки можно создать имя файла или каталога, которое при использовании будет транслироваться операционной системой в другое имя файла или каталога. Символьные ссылки — простой метод неявного разделения файлов или каталогов за счет создания перекрестных ссылок между различными каталогами в обычной иерархической структуре каталогов.

Диспетчер объектов реализует объект «символьная ссылка», который выполняет аналогичную функцию в отношении имен объектов в пространстве имен. Символьная ссылка может находиться в любом месте строки с именем объекта. Когда вызывающая программа ссылается на имя объекта «символьная ссылка», диспетчер просматривает пространство имен в поисках такого объекта. Далее он анализирует содержимое символьной ссылки и находит строку, которую надо подставить вместо ссылки. После этого начинается поиск другого объекта, соответствующего полученному имени.

Исполнительная система использует такие объекты при трансляции имен устройств в стиле MS-DOS во внутренние имена устройств Windows. Пользователь обращается к гибким и жестким дискам по именам A:, B:, C: и т. д. или к последовательным портам по именам COM1, COM2 и т. п. Подсистема Windows делает эти объекты «символьная ссылка» в защищенные глобальные данные, помещая их в каталог объектов \?? (в Windows 2000) или \GLOBAL?? (в Windows XP и Windows Server 2003).

Пространство имен сеанса

Windows NT изначально создавалась в расчете на регистрацию в системе одного интерактивного пользователя и выполнение лишь одного экземпляра любого из интерактивных приложений. Добавление Windows Terminal Services в Windows 2000 Server и поддержки быстрого переключения пользователей в Windows XP потребовало некоторых изменений в модели пространства имен диспетчера объектов для поддержки множества интерактивных пользователей одновременно. (Базовые сведения о службах терминала и сеансах см. в главе 1.)

Пользователь, зарегистрированный в консольном сеансе, получает доступ к *глобальному* пространству имен, которое является первым экземпляром пространства имен. Дополнительные сеансы получают свое (закрытое) представление пространства имен, называемое *локальным* пространством имен. Части пространства имен, локальные для каждого сеанса, включают \DosDevices, \Windows и \BaseNamedObjects. Формирование раздельных копий одних и тех же частей называется *созданием экземпляров* (instancing) пространства имен. Создание экземпляров каталога \DosDevices позволяет каждому пользователю обозначать сетевые дисковые устройства разными буквами и по-разному именовать такие объекты, как, например, последовательные порты. В Windows 2000 глобальный каталог \DosDevices называется \?? и является каталогом, на который указывает символьная ссылка \DosDevices, а локальные каталоги \DosDevices идентифицируются по идентификатору для сеанса сервера терминала. В Windows XP и более поздних операционных системах глобальный каталог \DosDevices называется \Global?? и является каталогом, на который указывает \DosDevices, а локальные каталоги \DosDevices определяются по идентификатору сеанса входа (logon session).

Win32k.sys создает в каталоге \Windows интерактивный объект WindowStation, \WinSta0. Среда Terminal Services может поддерживать несколько интерактивных пользователей, но для сохранения иллюзии доступа к предопределенному интерактивному объекту WindowStation в Windows каждому пользователю нужна собственная версия WinSta0. Наконец, в каталоге \BaseNamedObjects приложения и система создают разделяемые объекты, включая события, мьютексы и разделы. Если приложение, создающее именованный объект, запущено двумя пользователями, то в каждом сеансе нужна своя версия этого объекта, чтобы два экземпляра приложения не мешали друг другу, обращаясь к одному объекту.

Диспетчер объектов реализует локальное пространство имен, создавая закрытые версии трех каталогов, которые находятся в каталоге, сопоставленном с сеансом пользователя (\Sessions*X*, где *X* — идентификатор сеанса). Например, когда некое Windows-приложение во время удаленного сеанса номер 2 создает именованное событие, диспетчер объектов перенаправляет имя этого объекта из \BaseNamedObjects в \Sessions\2\BaseNamedObjects.

Все функции диспетчера объектов, связанные с управлением пространством имен, знают о локальных экземплярах каталогов и участвуют в под-

держании иллюзии того, что в удаленных сеансах используется то же пространство имен, что и в консольных. DLL-модули подсистемы Windows добавляют к именам, передаваемым Windows-приложениями, которые ссылаются на объекты в \DosDevices, префиксы \?? (например, C:\Windows превращается в \??\C:\Windows). Когда диспетчер объектов обнаруживает специальный префикс \??, предпринимаемые им действия зависят от версии Windows, но при этом он всегда полагается на поле *DeviceMap* в объекте «процесс», создаваемом исполнительной системой (executive process object) (EPROCESS, о котором пойдет речь в главе 6). Это поле указывает на структуру данных, разделяемую с другими процессами в том же сеансе. Поле *DosDevicesDirectory* структуры DeviceMap указывает на каталог диспетчера объектов, представляющий локальный \DosDevices процесса. Целевой каталог зависит от конкретной системы.

- Если системой является Windows 2000 и Terminal Services не установлены, поле *DosDevicesDirectory* в структуре DeviceMap процесса указывает на каталог \??, так как локальных пространств имен нет.
- Если системой является Windows 2000 и Terminal Services установлены, то, когда активным становится новый сеанс, система копирует все объекты из глобального каталога \?? в локальный для сеанса каталог \DosDevices, и поле *DosDevicesDirectory* структуры DeviceMap указывает на этот локальный каталог.
- В Windows XP и Windows Server 2003 система не копирует глобальные объекты в локальные каталоги DosDevices. Диспетчер объектов, встретив ссылку на \??, находит локальный для процесса каталог \DosDevices, используя поле *DosDevicesDirectory* структуры DeviceMap. Если нужного объекта в этом каталоге нет, он проверяет поле *DeviceMap* объекта «каталог» и, если это допустимо, ищет объект в каталоге, на который указывает поле *GlobalDosDevicesDirectory* структуры DeviceMap. Этим каталогом всегда является \Global??.

В определенных обстоятельствах приложениям, поддерживающим Terminal Services, нужен доступ к объектам в консольном сеансе, даже если сами приложения выполняются в удаленном сеансе. Это может понадобиться приложениям для синхронизации со своими экземплярами, выполняемыми в других удаленных или консольных сеансах. В таких случаях для доступа к глобальному пространству имен приложения могут использовать специальный префикс \Global, поддерживаемый диспетчером объектов. Так, объект \Global\ApplicationInitialized, открываемый приложением в сеансе номер 2, направляется вместо каталога \Sessions\2\BaseNamedObjects\ApplicationInitialized в каталог \BasedNamedObjects\ApplicationInitialized.

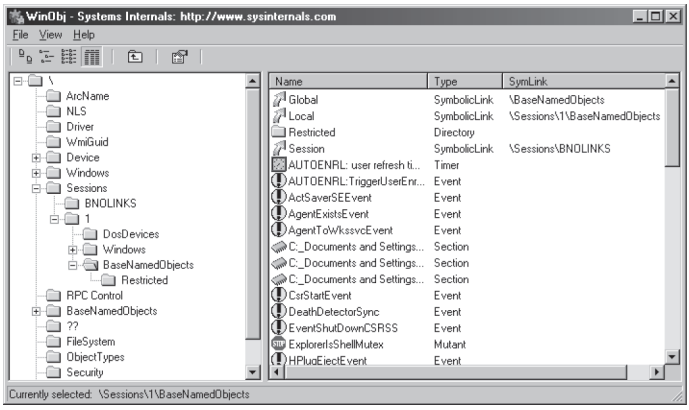
В Windows XP и Windows Server 2003 приложение, которому нужно обратиться к объекту в глобальном каталоге \DosDevices, не требуется использовать префикс \Global, если только этого объекта нет в локальном каталоге \DosDevices. Это вызвано тем, что диспетчер объектов автоматически ищет объект в глобальном каталоге, не найдя его в локальном. Однако при-

ложение, работающее в Windows 2000 с Terminal Services, должно всегда указывать префикс \Global для доступа к объектам в глобальном каталоге \DosDevices.

ЭКСПЕРИМЕНТ: просмотр экземпляров пространства имен

Вы можете увидеть, как диспетчер объектов создает экземпляры пространства имен, создав сеанс, отличный от консольного, и просмотрев таблицу описателей для процесса в этом сеансе. В Windows XP Home Edition или Windows XP Professional в системе, которая не входит в домен, отключите консольный сеанс [откройте меню Start (Пуск), щелкните Log Off (Выход из системы) и выберите Disconnect and Switch User (Смена пользователя) или нажмите комбинацию клавиш Windows+L]. Теперь войдите в систему под новой учетной записью. Если вы работаете с Windows 2000 Server, Advanced Server или Datacenter Server, запустите клиент Terminal Services, подключитесь к серверу и войдите в систему.

Войдя в систему в новом сеансе, запустите Winobj (*www.sysinternals.com*), щелкните каталог \Sessions и вы увидите подкаталог с числовым именем для каждого активного удаленного сеанса. Открыв один из таких каталогов, вы обнаружите подкаталоги \DosDevices, \Windows и \BaseNamedObjects, которые относятся к локальному пространству имен сеанса. Одно из таких локальных пространств имен показано на иллюстрации ниже.



Далее запустите Process Explorer и выберите какой-нибудь процесс в новом сеансе (вроде Explorer.exe). Просмотрите таблицу описателей, щелкнув View, Lower Pane View и Handles. Вы должны увидеть описатель \Windows\Windowstations\WinSta0 под \Sessions*n*, где *n* — идентификатор сеанса. Объекты с глобальными именами появятся в \Sessions*n*\BaseNamedObjects.

Синхронизация

Концепция *взаимоисключения* (mutual exclusion) является одной из ключевых при разработке операционных систем. Ее смысл в следующем: в каждый момент к конкретному ресурсу может обращаться один — и только один — поток. Взаимоисключение необходимо, когда ресурс не предназначен для разделения или когда такое разделение может иметь непредсказуемые последствия. Например, если бы два потока одновременно копировали данные в порт принтера, отпечатанный документ представлял бы собой нечитаемую мешанину. Аналогичным образом, если бы один поток считывал какой-то участок памяти, когда другой записывал бы туда данные, первый поток получил бы непредсказуемый набор данных. В общем случае доступные для записи ресурсы нельзя разделять без ограничений. Рис. 3-23 иллюстрирует, что происходит, когда два потока, выполняемые на разных процессорах, одновременно записывают данные в циклическую очередь.

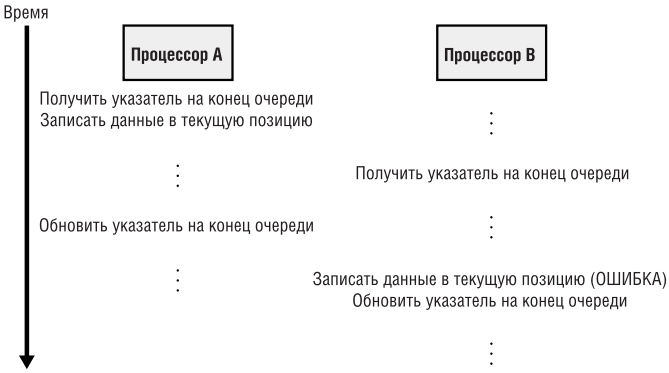


Рис. 3-23. *Некорректное разделение памяти*

Поскольку второй поток получил значение указателя на конец очереди до того, как первый поток завершил его обновление, второй вставил свои данные в то же место, что и первый. Таким образом, данные первого потока были перезаписаны другими данными, а один участок очереди остался пустым. Хотя рис. 3-23 иллюстрирует, что могло бы случиться в многопроцессорной системе, аналогичную ошибку было бы нельзя исключить и в однопроцессорной системе — при переключении контекста на второй поток до того, как первый поток успел бы обновить указатель на конец очереди.

Разделы кода, обращающиеся к неразделяемым ресурсам, называются *критическими секциями* (critical sections). В критической секции единовременно может выполняться только один поток. Пока один поток записывает в файл, обновляет базу данных или модифицирует общую переменную, доступ к этому ресурсу со стороны других потоков запрещен. Псевдокод, показанный на рис. 3-23, представляет собой критическую секцию, которая некорректно обращается к разделяемой структуре данных без взаимоисключения.

Взаимоисключение, важное для всех операционных систем, особенно значимо (и запутанно) в случае операционной системы с *жестко связанной симметричной мультипроцессорной обработкой* (tightly-coupled symmetric multiprocessing), например в Windows, в которой один и тот же системный код, выполняемый на нескольких процессорах одновременно, разделяет некоторые структуры данных, хранящиеся в глобальной памяти. В Windows поддержка механизмов, с помощью которых системный код может предотвратить одновременное изменение двумя потоками одной и той же структуры, возлагается на ядро. Оно предоставляет специальные примитивы взаимоисключения, используемые им и остальными компонентами исполнительной системы для синхронизации доступа к глобальным структурам данных.

Так как планировщик синхронизирует доступ к своим структурам данных при IRQL уровня «DPC/dispatch», ядро и исполнительная система не могут полагаться на механизмы синхронизации, которые могли бы привести к ошибке страницы или к перераспределению процессорного времени при IRQL уровня «DPC/dispatch» или выше (эти уровни также известны под названием «высокий IRQL»). Из следующих разделов вы узнаете, как ядро и исполнительная система используют взаимоисключение для защиты своих глобальных структур данных при высоком IRQL и какие механизмы синхронизации и взаимоисключения они применяют при низких уровнях IRQL (ниже «DPC/dispatch»).

Синхронизация ядра при высоком IRQL

Ядро должно гарантировать, что в каждый момент только один процессор выполняет код в критической секции. Критическими секциями ядра являются разделы кода, модифицирующие глобальные структуры данных, например базу данных диспетчера ядра или его очередь DPC. Операционная система не смогла бы корректно работать, если бы ядро не гарантировало взаимоисключающий доступ потоков к этим структурам данных.

В этом плане больше всего проблем с прерываниями. Так, в момент обновления ядром глобальной структуры данных может возникнуть прерывание, процедура обработки которого изменяет ту же структуру. В простых однопроцессорных системах развитие событий по такому сценарию исключается путем отключения всех прерываний на время доступа к глобальным данным, однако в ядре Windows реализовано более сложное решение. Перед использованием глобального ресурса ядро временно маскирует прерывания, обработчики которых используют тот же ресурс. Для этого ядро повышает IRQL процессора до самого высокого уровня, используемого любым потенциальным источником прерываний, который имеет доступ к глобальным данным. Например, прерывание на уровне «DPC/dispatch» приводит к запуску диспетчера ядра, использующего диспетчерскую базу данных. Следовательно, любая другая часть ядра, имеющая дело с этой базой данных, повышает IRQL до уровня «DPC/dispatch», маскируя прерывания того же уровня перед обращением к диспетчерской базе данных.

Эта стратегия хорошо работает в однопроцессорных системах, но не годится для многопроцессорных конфигураций. Повышение IRQL на одном из процессоров не исключает прерываний на другом процессоре, а ядро должно гарантировать взаимоисключающий доступ на всех процессорах.

Взаимоблокирующие операции

Простейшая форма механизмов синхронизации опирается на аппаратную поддержку безопасных операций над целыми значениями и выполнения сравнений в многопроцессорной среде. Сюда относятся такие функции, как *InterlockedIncrement*, *InterlockedDecrement*, *InterlockedExchange* и *InterlockedCompareExchange*. Скажем, функция *InterlockedDecrement*, использует префикс x86-инструкции *lock* (например, *lock xadd*) для блокировки многопроцессорной шины на время операции вычитания, чтобы другой процессор, модифицирующий тот же участок памяти, не смог выполнить свою операцию в момент между чтением исходных данных и записью их нового (меньшего) значения. Эта форма базовой синхронизации используется ядром и драйверами.

Спин-блокировки

Механизм, применяемый ядром для взаимоисключения в многопроцессорных системах, называется *спин-блокировкой* (spinlock). Спин-блокировка — это блокирующий примитив, сопоставленный с какой-либо глобальной структурой данных вроде очереди DPC (рис. 3-24).

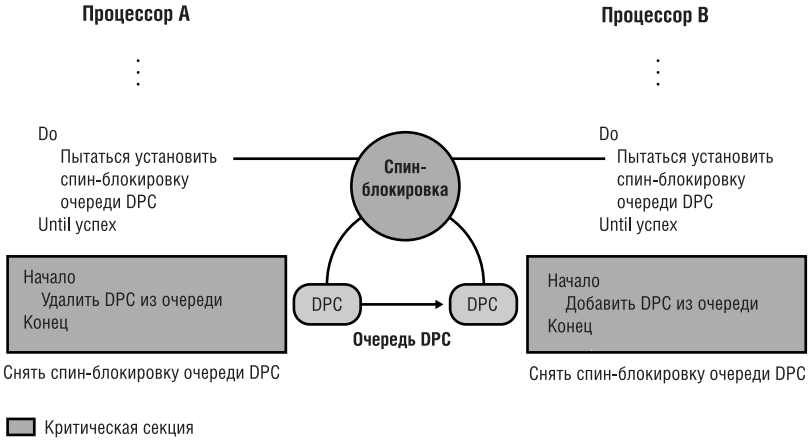


Рис. 3-24. *Применение спин-блокировки*

Перед входом в любую из критических секций, показанных на рис. 3-24, ядро должно установить спин-блокировку, связанную с защищенной очередью DPC. Если спин-блокировка пока занята, ядро продолжает попытки установить спин-блокировку до тех пор, пока не достигнет успеха. Термин получил такое название из-за поведения ядра (и соответственно процессора), которое «крутится» (spin) в цикле, повторяя попытки, пока не захватит блокировку.

Спин-блокировки, как и защищаемые ими структуры данных, находятся в глобальной памяти. Код для их установки и снятия написан на языке ассемблера для максимального быстродействия. Во многих архитектурах спин-блокировка реализуется аппаратно поддерживаемой командой test-and-set, которая проверяет значение переменной блокировки и устанавливает блокировку, выполняя всего одну атомарную команду. Это предотвращает захват блокировки вторым потоком в промежуток между проверкой переменной и установкой блокировки первым потоком.

Всем спин-блокировкам режима ядра в Windows назначен IRQL, всегда соответствующий уровню «DPC/dispatch» или выше. Поэтому, когда поток пытается установить спин-блокировку, все действия на этом или более низком уровне IRQL на данном процессоре прекращаются. Поскольку диспетчеризация потоков осуществляется при уровне «DPC/dispatch», поток, удерживающий спин-блокировку, никогда не вытесняется, так как данный IRQL маскирует механизмы диспетчеризации. Такая маскировка не дает прервать выполнение критической секции кода под защитой спин-блокировки и обеспечивает быстрое ее снятие. Спин-блокировки используются в ядре с большой осторожностью и устанавливаются на минимально возможное время.

ПРИМЕЧАНИЕ Поскольку IRQL — достаточно эффективный механизм синхронизации для однопроцессорных систем, функции установки и снятия спин-блокировки в однопроцессорных версиях HAL на самом деле просто повышают и понижают IRQL.

Ядро предоставляет доступ к спин-блокировкам другим компонентам исполнительной системы через набор функций ядра, включающий *KeAcquireSpinlock* и *KeReleaseSpinlock*. Например, драйверы устройств требуют спин-блокировки, чтобы система гарантировала единовременный доступ к регистрам устройства и другим глобальным структурам данных со стороны лишь одной части драйвера (и только с одного процессора). Спин-блокировка не предназначена для пользовательских программ — они должны оперировать объектами, которые рассматриваются в следующем разделе.

Спин-блокировки ядра накладывают ограничения на использующий их код. Как уже отмечалось, их IRQL всегда равен «DPC/dispatch», поэтому установивший спин-блокировку код может привести к краху системы, если попытается заставить планировщик выполнить операцию диспетчеризации или вызовет ошибку страницы.

Спин-блокировки с очередями

В некоторых ситуациях вместо стандартной спин-блокировки применяется особый тип спин-блокировки — с очередью (queued spinlock). Спин-блокировка с очередью лучше масштабируется в многопроцессорных системах, чем стандартная. Как правило, Windows использует лишь стандартные спин-блокировки, когда конкуренция за спин-блокировку ожидается низкой.

Спин-блокировка с очередью работает так: процессор, пытаясь установить такую спин-блокировку, которая в данный момент занята, ставит свой

идентификатор в очередь, сопоставленную с этой спин-блокировкой. Освободив спин-блокировку, удерживавший ее процессор передает блокировку тому процессору, чей идентификатор стоит в очереди первым. Между тем процессор, ожидающий занятую спин-блокировку, проверяет статус не самой спин-блокировки, а флага того процессора, чей идентификатор располагается в очереди прямо перед идентификатором ждущего процессора.

Тот факт, что спин-блокировка с очередью устанавливает флаги, а не глобальные блокировки, имеет два следствия. Во-первых, уменьшается интенсивный трафик, связанный с межпроцессорной синхронизацией. Во-вторых, вместо случайного выбора процессора из группы ожидающих спин-блокировку реализуется четкий порядок спин-блокировки по типу FIFO («первым вошел, первым вышел»). Такой порядок позволяет достичь более согласованной работы процессоров, использующих одну и ту же блокировку.

Windows определяет ряд глобальных спин-блокировок с очередями, сохраняя указатели на них в массиве, который содержится в блоке PCR (processor control region) каждого процессора. Глобальную спин-блокировку можно получить вызовом *KeAcquireQueuedSpinlock* с индексом в массиве PCR, по которому сохранен указатель на эту спин-блокировку. Количество глобальных спин-блокировок растет по мере появления новых версий операционной системы, и таблица их индексов публикуется в заголовочном файле Ntddk.h, поставляемом с DDK.

ЭКСПЕРИМЕНТ: просмотр глобальных спин-блокировок с очередями

Вы можете наблюдать за состоянием глобальных спин-блокировок с очередями, используя команду *!qlock* отладчика ядра. Эта команда имеет смысл лишь в многопроцессорной системе, так как в однопроцессорной версии HAL спин-блокировки не реализованы. В следующем примере (подготовленном в Windows 2000) спин-блокировка с очередью для базы данных диспетчера ядра удерживается процессором номер 1, а остальные спин-блокировки этого типа не затребованы (о базе данных диспетчера ядра см. главу 6).

```
kd> !qlocks
Key: O = Owner, 1-n = Wait order, blank = not owned/waiting,
     C = Corrupt

                      Processor Number
    Lock Name      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15

KE - Dispatcher       O
KE - Context Swap
MM - PFN
MM - System Space
CC - Vacb
CC - Master
```

Внутристековые спин-блокировки с очередями

Помимо статических спин-блокировок с очередями, определяемых глобально, ядра Windows XP и Windows Server 2003 поддерживают динамически создаваемые спин-блокировки с очередями. Для их создания предназначены функции *KeAcquireInStackQueuedSpinlock* и *KeReleaseInStackQueuedSpinlock*. Этот тип блокировок используется несколькими компонентами, в том числе диспетчером кэша, диспетчером пулов исполнительной системы (executive pool manager) и NTFS. Упомянутые функции документированы в DDK для сторонних разработчиков драйверов.

KeAcquireInStackQueuedSpinlock принимает указатель на структуру данных спин-блокировки и описатель очереди спин-блокировки. Этот описатель в действительности является структурой данных, в которой ядро хранит информацию о состоянии блокировки, в частности сведения о владельце блокировки и об очереди процессоров, ожидающих освобождения этой блокировки.

Взаимоблокирующие операции в исполнительной системе

Ядро предоставляет ряд функций синхронизации, использующих спин-блокировки для более сложных операций, например для добавления и удаления элементов из одно- и двунаправленных связанных списков. К таким функциям, в частности, относятся ExInterlockedPopEntryList и ExInterlockedPushEntryList (для однонаправленных связанных списков), ExInterlockedInsertHeadList и ExInterlockedRemoveHeadList (для двунаправленных связанных списков). Все эти функции требуют передачи стандартной спин-блокировки в качестве параметра и интенсивно используются в ядре и драйверах устройств.

Синхронизация при низком IRQL

Компоненты исполнительной системы вне ядра также нуждаются в синхронизации доступа к глобальным структурам данных в многопроцессорной среде. Например, у диспетчера памяти есть только одна база данных блоков страниц. Обращение к ней осуществляется как к глобальной структуре данных, и драйверам устройств необходима гарантия получения монопольного доступа к своим устройствам. Вызывая функции ядра, исполнительная система может создать спин-блокировку, установить ее и снять.

Однако спин-блокировка лишь частично удовлетворяет потребности исполнительной системы в синхронизации. Поскольку спин-блокировка означает фактическую остановку процессора, она применяется только при двух условиях:

- требуется непродолжительное обращение к защищенным ресурсам без сложного взаимодействия с другим кодом;
- код критической секции нельзя выгрузить в страничный файл, он не ссылается на данные в подкачиваемой памяти, не вызывает внешние процедуры (включая системные сервисы) и не генерирует прерывания или исключения.

Эти противоречащие друг другу ограничения нельзя соблюсти одновременно ни при каких обстоятельствах. Более того, кроме взаимоисключения, исполнительная система должна выполнять и другие алгоритмы синхронизации, а также предоставлять механизмы синхронизации пользовательскому режиму.

Существует несколько дополнительных механизмов синхронизации, применяемых, когда спин-блокировки не годятся:

- объекты диспетчера ядра (kernel dispatcher objects);
- быстрые мьютексы (fast mutexes) и защищенные мьютексы (guarded mutexes);
- блокировки с заталкиванием указателя (push locks);
- ресурсы исполнительной системы (executive resources).

В таблице 3-9 кратко сравниваются возможности этих механизмов и их взаимосвязь с доставкой APC режима ядра.

Таблица 3-9. *Механизмы синхронизации режима ядра*

	Предоставляется драйверам устройств	Отключает обычные APC режима ядра доступ	Отключает специальные APC режима ядра	Поддерживает разделяемый и монопольный захват	Поддерживает рекурсивный захват
Мьютексы диспетчера ядра	Да	Да	Нет	Да	Нет
Семафоры диспетчера ядра	Да	Нет	Нет	Нет	Нет
Быстрые мьютексы	Да	Да	Да	Нет	Нет
Защищенные мьютексы	Нет	Да	Да	Нет	Нет
Блокировки с заталкиванием указателя	Нет	Нет	Нет	Нет	Да
Ресурсы исполнительной системы	Да	Да	Нет	Да	Да

Объекты диспетчера ядра

Ядро предоставляет исполнительной системе дополнительные механизмы синхронизации в форме объектов, в совокупности известных как объекты диспетчера ядра. Синхронизирующие объекты, видимые из пользовательского режима, берут свое начало именно от этих объектов диспетчера ядра. Каждый синхронизирующий объект, видимый из пользовательского режима, инкапсулирует минимум один объект диспетчера ядра. Семантика синхронизации исполнительной системы доступна программистам через Windows-функции *WaitForSingleObject* и *WaitForMultipleObjects*, реализуемые подсистемой Windows на основе аналогичных системных сервисов, предоставляемых диспетчером объектов. Поток в Windows-приложении можно синхронизировать

по таким Windows-объектам, как процесс, поток, событие, семафор, мьютекс, ожидаемый таймер, порт завершения ввода-вывода или файл.

Еще один тип синхронизирующих объектов исполнительной системы назван (без особой на то причины) *ресурсами исполнительной системы* (executive resources). Эти ресурсы обеспечивают как монопольный доступ (по аналогии с мьютексами), так и разделяемый доступ для чтения (когда несколько потоков-«читателей» обращается к одной структуре только для чтения). Однако они доступны лишь коду режима ядра, а значит, недоступны через Windows API. Ресурсы исполнительной системы являются не объектами диспетчера ядра, а скорее структурами данных, память для которых выделяется прямо из неподкачиваемого пула, имеющего свои специализированные сервисы для инициализации, блокировки, освобождения, запроса и ожидания. Структура ресурсов исполнительной системы определена в Ntddk.h, а соответствующие процедуры описаны в DDK.

В остальных подразделах мы детально обсудим, как реализуется ожидание на объектах диспетчера ядра.

Ожидание на объектах диспетчера ядра Поток синхронизируется с объектом диспетчера ядра, ожидая освобождения его описателя. При этом ядро приостанавливает поток и соответственно меняет состояние диспетчера, как показано на рис. 3-25. Ядро удаляет поток из очереди готовых к выполнению потоков и перестает учитывать его в планировании.

ПРИМЕЧАНИЕ На рис. 3-25 показана схема перехода состояний с выделением состояний «готов» (ready), «ожидает» (waiting) и «выполняется» (running) — они относятся к ожиданию на объектах. Прочие состояния описываются в главе 6.

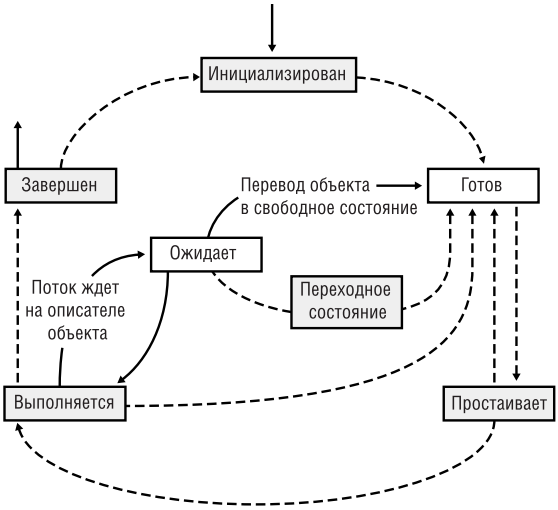


Рис. 3-25. *Ожидание на объекте диспетчера ядра*

В любой момент синхронизирующий объект находится в одном из двух состояний: *свободном* (signaled) или *занятом* (nonsignaled). Для синхронизации с объектом поток вызывает один из системных сервисов ожидания, предоставляемых диспетчером объектов, и передает описатель этого объекта. Поток может ожидать на одном или нескольких объектах, а также указать, что ожидание следует прекратить, если объект (или объекты) не освободился в течение определенного времени. Всякий раз, когда ядро переводит объект в свободное состояние, функция *KiWaitTest* ядра проверяет, ждут ли этот объект какие-нибудь потоки и не ждут ли они каких-либо других объектов. Если да, ядро выводит один или более потоков из состояния ожидания, после чего их выполнение может быть продолжено.

Взаимосвязь синхронизации с диспетчеризацией потоков иллюстрирует следующий пример с использованием объекта «событие».

- Поток пользовательского режима ждет на описателе объекта «событие» (т. е. ждет перехода этого объекта в свободное состояние).
- Ядро изменяет состояние потока с «готов» на «ожидает» и добавляет его в список потоков, ждущих объект «событие».
- Другой поток устанавливает объект «событие».
- Ядро просматривает список потоков, ожидающих этот объект. Если условия ожидания какого-либо потока выполнены (см. примечание ниже), ядро переводит его из состояния «ожидает» в состояние «готов». Если это поток с динамическим приоритетом, ядро может повысить его приоритет для выполнения.
- Поскольку новый поток теперь готов к выполнению, происходит перераспределение процессорного времени. Если при этом диспетчер обнаружит, что приоритет выполняемого потока ниже, чем приоритет потока, только что перешедшего в состояние «готов», он вытеснит поток с более низким приоритетом и выдаст программное прерывание для инициации переключения контекста на поток с более высоким приоритетом.
- Если в данный момент вытеснение невозможно ни на одном из процессоров, диспетчер включает поток в свою очередь потоков, готовых к выполнению.

ПРИМЕЧАНИЕ Некоторые потоки могут ждать более одного объекта, и в таком случае их ожидание продолжается.

Условия перехода объектов в свободное состояние Эти условия различны для разных объектов. Например, объект «поток» находится в занятом состоянии в течение всего срока своей жизни и переводится ядром в свободное состояние лишь при завершении. Аналогичным образом, ядро переводит объект «процесс» в свободное состояние в момент завершения последнего потока процесса. Но такой объект, как таймер, переводится в свободное состояние по истечении заданного времени.

Выбирая механизм синхронизации, вы должны учитывать в своей программе поведение синхронизирующих объектов. В таблице 3-10 показано, когда переходят в свободное состояние синхронизирующие объекты различных типов.

Таблица 3-10. *Условия освобождения различных синхронизирующих объектов*

Тип объекта	Переводится в свободное состояние при:	Действие на ожидающие потоки
Процесс	Завершении последнего потока	Все потоки освобождаются
Поток	Завершении данного потока	Все потоки освобождаются
Файл	Завершении операции ввода-вывода	Все потоки освобождаются
Событие (уведомляющего типа)	Установке события потоком	Все потоки освобождаются
Событие (синхронизирующего типа)	Установке события потоком	Освобождается один поток, и объект «событие» сбрасывается
Семафор	Уменьшении счетчика семафора на 1	Освобождается один поток
Таймер (уведомляющего типа)	Истечении заданного времени или наступлении очередного момента срабатывания	Все потоки освобождаются
Таймер (синхронизирующего типа)	Истечении заданного времени или наступлении очередного момента срабатывания	Освобождается один поток
Мьютекс	Освобождении объекта «мьютекс» потоком	Освобождается один поток
Очередь	Добавлении элемента в очередь	Освобождается один поток

Когда объект переводится в свободное состояние, ожидающие его потоки обычно немедленно выходят из ждущего состояния. Однако, как показано на рис. 3-26, некоторые объекты диспетчера ядра и системные события ведут себя иначе.

Например, объект «событие уведомления» — в Windows API он называется событием со сбросом вручную (manual reset event) — используется для уведомления о каком-либо событии. Когда этот объект переводится в свободное состояние, все потоки, ожидающие его, освобождаются. Исключением является тот поток, который ждет сразу несколько объектов: он может продолжать ожидание, пока не освободятся дополнительные объекты.

В отличие от события мьютекс предусматривает возможность владения. Этот объект используется для взаимоисключающего доступа к ресурсу, поэтому единовременно только один поток может владеть мьютексом. При освобождении мьютекса ядро переводит его в свободное состояние и выбирает для выполнения один из ожидающих потоков. Выбранный ядром поток захватывает мьютекс, а остальные потоки остаются в ожидании.

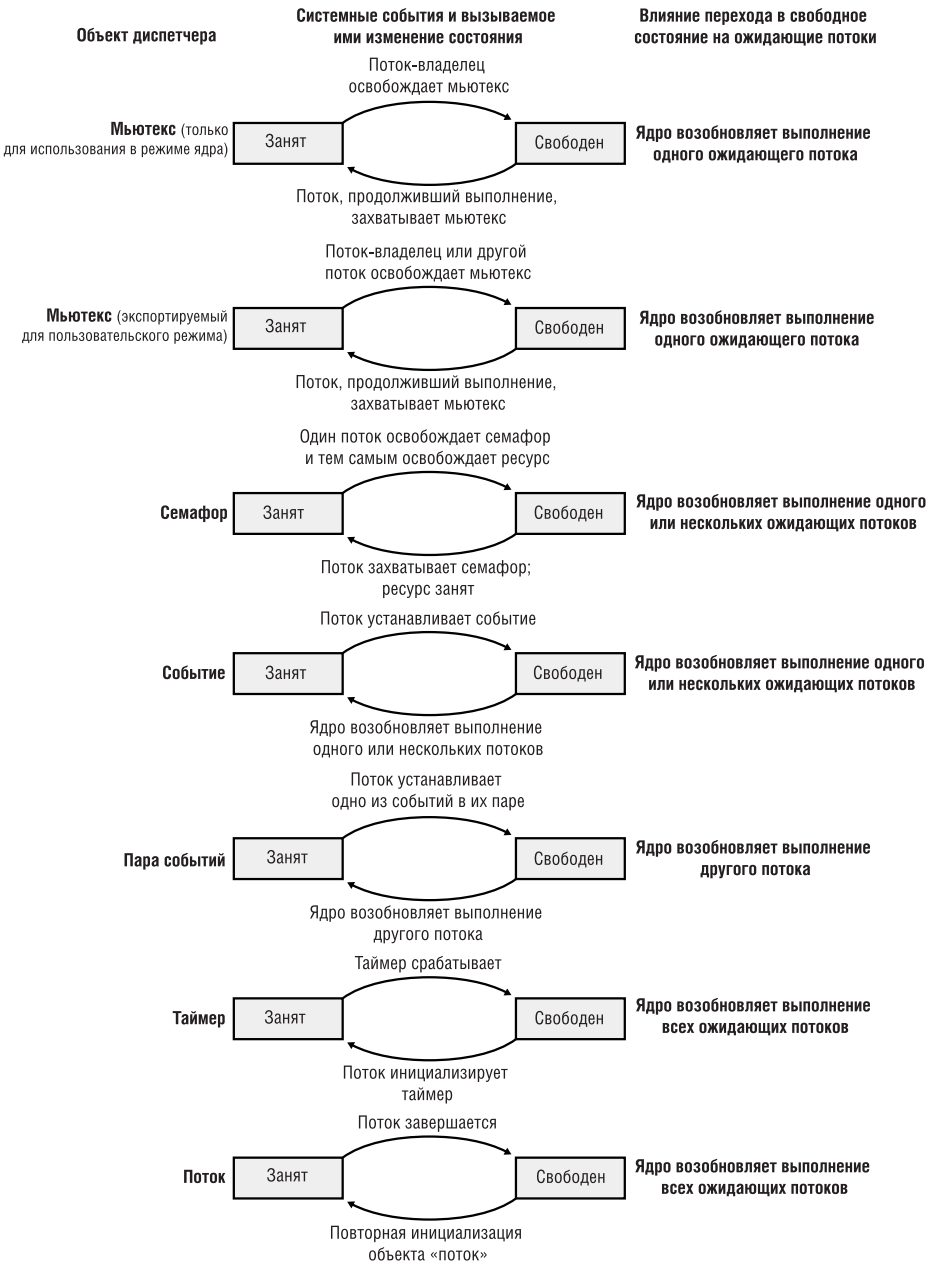


Рис. 3-26. *Некоторые объекты диспетчера ядра*

События с ключом и критические секции

Синхронизирующий объект, впервые появившийся в Windows XP и названный *событием с ключом* (keyed event), заслуживает особого упоминания. Он помогает процессам справляться с нехваткой памяти при использовании критических секций. Это недокументированное событие позволяет потоку указать «ключ» в следующей ситуации: данный поток должен пробуждаться, когда другой поток того же процесса освобождает событие с тем же ключом.

Windows-процессы часто используют функции критических секций — *EnterCriticalSection* и *LeaveCriticalSection* — для синхронизации доступа потоков к личным ресурсам процесса. Вызовы этих функций эффективнее прямого обращения к объектам «мьютекс», так как в отсутствие конкуренции они не заставляют переходить в режим ядра. При наличии конкуренции *EnterCriticalSection* динамически создает объект «событие», и поток, которому нужно захватить критическую секцию, ждет, когда поток, владеющий этой секцией, освободит ее вызовом *LeaveCriticalSection*.

Если создать объект «событие» для критической секции не удалось из-за нехватки системной памяти, *EnterCriticalSection* использует глобальное событие с ключом — *CritSecOutOfMemoryEvent* (в каталоге \Kernel пространства имен диспетчера объектов). Если *EnterCriticalSection* вынуждена задействовать *CritSecOutOfMemoryEvent* вместо стандартного события, поток, ждущий критическую секцию, использует адрес этой секции как ключ. Это обеспечивает корректную работу функций критических секций даже в условиях временной нехватки памяти.

Мы не ставили себе задачу исчерпывающе описать все объекты исполнительной системы, а лишь хотели дать представление об их базовой функциональности и механизмах синхронизации. Об использовании этих объектов в Windows-программах см. справочную документацию Windows или четвертое издание книги Джеффри Рихтера «Windows для профессионалов».

Структуры данных Учет ожидающих потоков и их объектов ожидания базируется на двух ключевых структурах данных: *заголовках диспетчера* (dispatcher headers) и *блоках ожидания* (wait blocks). Обе эти структуры определены в Ntddk.h, заголовочном файле DDK. Для удобства мы воспроизводим здесь эти определения.

```
typedef struct _DISPATCHER_HEADER {
   UCHAR Type;
   UCHAR Absolute;
   UCHAR Size;
   UCHAR Inserted;
   LONG SignalState;
   LIST_ENTRY WaitListHead;
} DISPATCHER_HEADER;
```

```
typedef struct _KWAIT_BLOCK {
   LIST_ENTRY WaitListEntry;
   struct _KTHREAD *RESTRICTED_POINTER Thread;
   PVOID Object;
   struct _KWAIT_BLOCK *RESTRICTED_POINTER NextWaitBlock;
   USHORT WaitKey;
   USHORT WaitType;
} KWAIT_BLOCK, *PKWAIT_BLOCK, *RESTRICTED_POINTER PRKWAIT_BLOCK;
```

Заголовок диспетчера содержит тип объекта, информацию о состоянии (занят/свободен) и список потоков, ожидающих этот объект. У каждого ждущего потока есть список блоков ожидания, где перечислены ожидаемые потоком объекты, а у каждого объекта диспетчера ядра — список блоков ожидания, где перечислены ожидающие его потоки. Этот список ведется так, что при освобождении объекта диспетчера ядро может быстро определить, кто ожидает данный объект. В блоке ожидания имеются указатели на объект ожидания, ожидающий поток и на следующий блок ожидания (если поток ждет более одного объекта). Он также регистрирует тип ожидания («любой» или «все») и позицию соответствующего элемента в таблице описателей, переданную потоком в функцию *WaitForMultipleObjects* (позиция 0 — если поток ожидает лишь один объект).

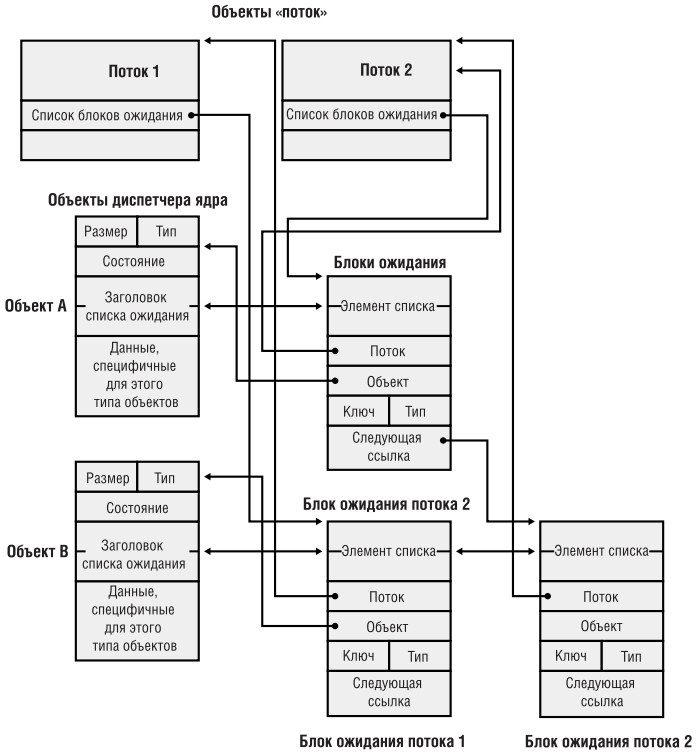


Рис. 3-27. *Структуры данных, необходимые для поддержки ожидания*

На рис. 3-27 показана связь объектов диспетчера ядра с блоками ожидания потоков. В данном примере поток 1 ждет объект B, а поток 2 — объекты A и B. Если объект A освободится, поток 2 не сможет возобновить свое выполнение, так как ядро обнаружит, что он ждет и другой объект. С другой стороны, при освобождении объекта B ядро сразу же подготовит поток 1 к выполнению, поскольку он не ждет никакие другие объекты.

ЭКСПЕРИМЕНТ: просмотр очередей ожидания

Хотя многие утилиты просмотра процессов умеют определять, находится ли поток в состоянии ожидания (отмечая в этом случае и тип ожидания), список объектов, ожидаемых потоком, можно увидеть только с помощью команды *!process* отладчика ядра. Например, следующий фрагмент вывода команды *!process* показывает, что поток ждет на объекте-событии.

```
kd> !process
...
        THREAD 8a12a328  Cid 0bb8.0d50  Teb: 7ffdd000
Win32Thread: e7c9aeb0 WAIT: (WrUserRequest) UserMode
Non-Alertable
            8a21bf58  SynchronizationEvent
```

Команда *dt* позволяет интерпретировать содержимое заголовка диспетчера, например, такого объекта:

```
kd> dt nt!_dispatcher_header 8a21bf58
nt!_DISPATCHER_HEADER
   +0x000 Type             : 0x1 ''
   +0x001 Absolute         : 0 ''
   +0x002 Size             : 0x4 ''
   +0x003 Inserted         : 0 ''
   +0x004 SignalState      : 0
   +0x008 WaitListHead     : _LIST_ENTRY [ 0x8a12a398 -
                             0x8a12a398 ]
```

Эти данные позволяют нам убедиться в отсутствии других потоков, ожидающих данный объект, поскольку указатели начала и конца списка ожидания указывают на одно и то же место (на один блок ожидания). Копия блока ожидания (по адресу 0x8a12a398) дает следующее:

```
kd> dt nt!_kwait_block 0x8a12a398
nt!_KWAIT_BLOCK
   +0x000 WaitListEntry    : _LIST_ENTRY [ 0x8a21bf60 -
                             0x8a21bf60 ]
   +0x008 Thread           : 0x8a12a328
   +0x00c Object           : 0x8a21bf58
   +0x010 NextWaitBlock    : 0x8a12a398
   +0x014 WaitKey          : 0
   +0x016 WaitType         : 1
```

Если в списке ожидания более одного элемента, вы можете выполнить ту же команду со вторым указателем в поле *WaitListEntry* каждого блока ожидания (команду *!thread* применительно к указателю потока в блоке ожидания) для прохода по списку и просмотра других потоков, ждущих данный объект.

Быстрые и защищенные мьютексы

Быстрые мьютексы (fast mutexes), также известные как мьютексы исполнительной системы, обычно обеспечивают более высокую производительность, чем объекты «мьютекс». Почему? Дело в том, что быстрые мьютексы, хоть и построены на объектах событий диспетчера, в отсутствие конкуренции не требуют ожидания объекта «событие» (и соответственно спин-блокировок, на которых основан этот объект). Эти преимущества особенно ярко проявляются в многопроцессорной среде. Быстрые мьютексы широко используются в ядре и драйверах устройств.

Однако быстрые мьютексы годятся, только если можно отключить доставку обычных APC режима ядра. В исполнительной системе определены две функции для захвата быстрых мьютексов: *ExAcquireFastMutex* и *ExAcquireFastMutexUnsafe*. Первая функция блокирует доставку всех APC, повышая IRQL процессора до уровня APC_LEVEL, а вторая — ожидает вызова при уже отключенной доставке обычных APC режима ядра (такое отключение возможно повышением IRQL до уровня «APC» или вызовом *KeEnterCriticalRegion*). Другое ограничение быстрых мьютексов заключается в том, что их нельзя захватывать рекурсивно, как объекты «мьютекс».

Защищенные мьютексы (guarded mutexes) — новшество Windows Server 2003; они почти идентичны быстрым мьютексам (хотя внутренне используют другой синхронизирующий объект, KGATE). Захватить защищенные мьютексы можно вызовом функции *KeAcquireGuardedMutex*, отключающей доставку всех APC режима ядра через *KeEnterGuardedRegion*, а не *KeEnterCriticalRegion*, которая на самом деле отключает только обычные APC режима ядра. Защищенные мьютексы недоступны вне ядра и используются в основном диспетчером памяти для защиты глобальных операций вроде создания страничных файлов, удаления определенных типов разделов общей памяти и расширения пула подкачиваемой памяти. (Подробнее о диспетчере памяти см. главу 7.)

Ресурсы исполнительной системы

Ресурсы исполнительной системы (executive resources) — это механизм синхронизации, который поддерживает разделяемый (совместный) и монопольный доступ и по аналогии с быстрыми мьютексами требует предварительного отлючения доставки обычных APC режима ядра. Они основаны на объектах диспетчера, которые используются только при наличии конкуренции. Ресурсы исполнительной системы широко применяются во всей системе, особенно в драйверах файловой системы.

Потоки, которым нужно захватить какой-либо ресурс для совместного доступа, ждут на семафоре, сопоставленном с этим ресурсом, а потоки, которым требуется захватить ресурс для монопольного доступа, — на событии. Семафор с неограниченным счетчиком применяется потому, что в первом случае можно пробудить все ждущие потоки и предоставить им доступ к ресурсу, как только этот семафор перейдет в свободное состояние (ресурс будет освобожден потоком, захватившим его в монопольное владение). Когда потоку нужен монопольный доступ к занятому на данный момент ресурсу, он ждет на синхронизирующем объекте «событие», так как при освобождении события пробуждается только один из ожидающих потоков.

Для захвата ресурсов предназначен целый ряд функций: *ExAcquireResourceSharedLite*, *ExAcquireResourceExclusiveLite*, *ExAcquireSharedStarveExclusive*, *ExAcquireWaitForExclusive* и *ExTryToAcquireResourceExclusiveLite*. Эти функции документированы в DDK.

ЭКСПЕРИМЕНТ: перечисление захваченных ресурсов исполнительной системы

Команда *!locks* отладчика ядра ищет в пуле подкачиваемой памяти объекты ресурсов исполнительной системы и выводит их состояние. По умолчанию эта команда перечисляет только захваченные на данный момент ресурсы, но ключ *–d* позволяет перечислять все ресурсы исполнительной системы. Вот фрагмент вывода этой команды:

```
lkd> !locks
**** DUMP OF ALL RESOURCE OBJECTS ****
KD: Scanning for held locks.

Resource @ nt!MmSystemWsLock (0x805439a0)    Exclusively owned
    Contention Count = 123
     Threads: 89b36020-01<*>
KD: Scanning for held locks.....................................
.............................................................
.............................................................
............

Resource @ 0x89da1a68    Shared 1 owning threads
     Threads: 8a4cb533-01<*> *** Actual Thread 8a4cb530
```

Заметьте, что счетчик конкурирующих потоков (contention count), извлекаемый из структуры ресурса, фиксирует, сколько раз потоки пытались захватить данный ресурс и были вынуждены переходить в состояние ожидания из-за того, что он уже занят.

Для изучения деталей конкретного объекта ресурса (в частности, кто владеет ресурсом и кто ждет его освобождения) укажите ключ *–v* и адрес ресурса:

```
lkd> !locks -v 0x805439a0
```

```
Resource @ nt!MmSystemWsLock (0x805439a0)    Exclusively owned
    Contention Count = 123
     Threads: 89b36020-01<*>

     THREAD 89b36020  Cid 0e98.0bd4  Teb: 7ffd9000 Win32Thread:
e2bcc538 RUNNING on processor 0
     Not impersonating
     DeviceMap                 e1df7d18
     Owning Process            8999d020
     Wait Start TickCount      492582         Elapsed Ticks: 15
     Context Switch Count      532                 LargeStack
     UserTime                  00:00:01.0462
     KernelTime                00:00:00.0320
     Start Address 0x77e7d342
     Win32 Start Address 0x0101f1d0
     Stack Init a9d20000 Current a9d1fd44 Base a9d20000
     Limit a9d1d000 Call 0
     Priority 11 BasePriority 8 PriorityDecrement 2
     DecrementCount 16
Unable to get context for thread running on processor 0,
HRESULT 0x80004001
```

Блокировки с заталкиванием указателя

Блокировки с заталкиванием указателя (push locks), впервые появившиеся в Windows XP, являются еще одним оптимизированным механизмом синхронизации, который основан на объекте «событие» (в Windows Server 2003 такие блокировки базируются на внутреннем синхронизирующем объекте KGATE) и подобно быстрым мьютексам заставляет ждать этот объект только при наличии конкуренции. Такие блокировки имеют преимущества над быстрыми мьютексами, так как их можно захватывать как в разделяемом, так и в монопольном режиме. Они не документированы и не экспортируются ядром, так как зарезервированы для использования самой операционной системой.

Существует два типа блокировок с заталкиванием указателя: обычный и с поддержкой кэша (cache aware). Первый тип занимает в памяти тот же объем, что и указатель (4 байта в 32-разрядных системах и 8 байтов в 64-разрядных). Когда поток захватывает обычную блокировку с заталкиванием указателя, код этой блокировки помечает ее как занятую, если она на данный момент свободна. Если блокировка захвачена для монопольного доступа или если потоку нужно захватить ее монопольно, а она уже захвачена для разделяемого доступа, ее код создает в стеке потока блок ожидания, инициализирует объект «событие» в этом блоке и добавляет последний в список ожидания, сопоставленный с блокировкой. Как только блокировка освобождается, ее код пробуждает ждущий поток (если таковой имеется), освобождая событие в блоке ожидания потока.

Второй тип создает обычную блокировку с заталкиванием указателя для каждого процессора в системе и сопоставляет ее с блокировкой с заталкиванием указателя, поддерживающей кэш. Когда потоку нужно захватить такую блокировку, он просто захватывает обычную блокировку, созданную для текущего процессора в соответствующем режиме доступа.

Подобные блокировки используются, в том числе, диспетчером объектов, когда возникает необходимость в защите глобальных структур данных и дескрипторов защиты объектов, а также диспетчером памяти для защиты структур данных AWE.

Обнаружение взаимоблокировки с помощью Driver Verifier

Взаимоблокировка (deadlock) — это проблема синхронизации, возникающая, когда два потока или процессора удерживают ресурсы, нужные другому, и ни один из них не отдает их. Такая ситуация может приводить к зависанию системы или процесса. Утилита Driver Verifier, описываемая в главах 7 и 9, позволяет проверять возможность взаимоблокировки, в том числе на спин-блокировках, быстрых и обычных мьютексах. О том, как пользоваться Driver Verifier для анализа зависания системы, см. главу 14.

Системные рабочие потоки

При инициализации Windows создает несколько потоков в процессе System, которые называются *системными рабочими потоками* (system worker threads). Они предназначены исключительно для выполнения работы по поручению других потоков. Во многих случаях потоки, выполняемые на уровне «DPC/dispatch», нуждаются в вызове таких функций, которые могут быть вызваны только при более низком IRQL. Например, DPC-процедуре, выполняемой в контексте произвольного потока при IRQL уровня «DPC/dispatch» (DPC может узурпировать любой поток в системе), нужно обратиться к пулу подкачиваемой памяти или ждать на объекте диспетчера для синхронизации с потоком какого-либо приложения. Поскольку DPC-процедура не может понизить IRQL, она должна передать свою задачу потоку, который сможет выполнить ее при IRQL ниже уровня «DPC/dispatch».

Некоторые драйверы устройств и компоненты исполнительной системы создают собственные потоки для обработки данных на уровне «passive», но большинство вместо этого использует системные рабочие потоки, что помогает избежать слишком частого переключения потоков и чрезмерной нагрузки на память из-за диспетчеризации дополнительных потоков. Драйвер устройства или компонент исполнительной системы запрашивает сервисы системных рабочих потоков через функцию исполнительной системы *ExQueueWorkItem* или *IoQueueWorkItem*. Эти функции помещают *рабочий элемент* (work item) в специальную очередь, проверяемую системными рабочими потоками (см. раздел «Порты завершения ввода-вывода» главы 9).

Рабочий элемент включает указатель на процедуру и параметр, передаваемый потоком этой процедуре при обработке рабочего элемента. Процедура реализуется драйвером устройства или компонентом исполнительной системы, выполняемым на уровне «passive».

Например, DPC-процедура, которая должна ждать на объекте диспетчера, может инициализировать рабочий элемент, который указывает на процедуру в драйвере, ждущем на объекте диспетчера, и, возможно, на указатель на объект. На каком-то этапе системный рабочий поток извлекает из своей очереди рабочий элемент и выполняет процедуру драйвера. После ее выполнения системный рабочий поток проверяет, нет ли еще рабочих элементов, подлежащих обработке. Если нет, системный рабочий поток блокируется, пока в очередь не будет помещен новый рабочий элемент. Выполнение DPC-процедуры может и не закончиться в ходе обработки ее рабочего элемента системным рабочим потоком. (В однопроцессорной системе выполнение этой процедуры всегда завершается до обработки ее рабочего элемента, так как на уровне IRQL «DPC/dispatch» потоки не планируются.)

Существует три типа системных рабочих потоков:

- *отложенные* (delayed worker threads) — выполняются с приоритетом 12, обрабатывают некритичные по времени рабочие элементы и допускают выгрузку своего стека в страничный файл на время ожидания рабочих элементов;
- *критичные* (critical worker threads) — выполняются с приоритетом 13, обрабатывают критичные по времени рабочие элементы. В Windows Server их стек всегда находится только в физической памяти;
- *гиперкритичный* (hypercritical worker thread) — единственный поток, выполняемый с приоритетом 15. Его стек тоже всегда находится в памяти. Диспетчер процессов использует гиперкритичные по времени рабочие элементы для выполнения функции, освобождающей завершенные потоки.

Число отложенных и критичных системных рабочих потоков, создаваемых функцией исполнительной системы *ExpWorkerInitialization*, которая вызывается на ранних стадиях процесса загрузки, зависит от объема памяти в системе и от того, является ли система сервером. В таблице 3-11 показано количество потоков, изначально создаваемых в системах с различной конфигурацией. Вы можете указать *ExpInitializeWorker* создать дополнительно до 16 отложенных и 16 критичных системных рабочих потоков. Для этого используйте параметры AdditionalDelayedWorkerThreads и AdditionalCriticalWorkerThreads в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\ Session Manager\Executive.

Таблица 3-11. *Начальное количество системных рабочих потоков*

	Windows 2000	Windows 2000 Server	Windows XP и Windows Server 2003
Отложенный	3	3	7
Критичный	5	10	5
Гиперкритичный	1	1	1

Исполнительная система старается балансировать число критичных системных рабочих потоков в соответствии с текущей рабочей нагрузкой. Каждую секунду функция исполнительной системы *ExpWorkerThreadBalanceManager* проверяет, надо ли создавать новый критичный рабочий поток. Кстати, критичный рабочий поток, создаваемый функцией *ExpWorkerThreadBalanceManager*, называется *динамическим* (dynamic worker thread). Для создания такого потока должны быть выполнены следующие условия.

- Очередь критичных рабочих элементов не должна быть пустой.
- Число неактивных критичных потоков (блокированных в ожидании рабочих элементов или на объектах диспетчера при выполнении рабочей процедуры) должно быть меньше количества процессоров в системе.
- В системе должно быть менее 16 динамических рабочих потоков.

Динамические потоки завершаются через 10 минут пребывания в неактивном состоянии. В зависимости от рабочей нагрузки исполнительная система может создавать до 16 таких потоков.

ЭКСПЕРИМЕНТ: просмотр списка системных рабочих потоков

Для получения списка системных рабочих потоков, сгруппированных по типу, используйте команду *!exqueue* отладчика ядра:

```
kd> !exqueue
Dumping ExWorkerQueue: 8046A5C0

**** Critical WorkQueue( current = 0 maximum = 1 )
THREAD 818a2d40  Cid 8.c   Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a2ac0  Cid 8.10  Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a2840  Cid 8.14  Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a25c0  Cid 8.18  Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a2340  Cid 8.1c  Teb: 00000000  Win32Thread: 00000000 WAIT

**** Delayed WorkQueue( current = 0 maximum = 1 )
THREAD 818a20c0  Cid 8.20  Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a1020  Cid 8.24  Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a1da0  Cid 8.28  Teb: 00000000  Win32Thread: 00000000 WAIT

**** HyperCritical WorkQueue( current = 0 maximum = 1 )
THREAD 818a1b20  Cid 8.2c  Teb: 00000000  Win32Thread: 00000000 WAIT
```

Глобальные флаги Windows

Windows поддерживает набор флагов, который хранится в общесистемной глобальной переменной *NtGlobalFlag*, предназначенной для отладки, трассировки и контроля операционной системы. При загрузке системы переменная *NtGlobalFlag* инициализируется значением параметра GlobalFlag из раздела реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager. По умолчанию его значение равно 0, и в системах с обычной конфигурацией глобальные флаги обычно не используются. Кроме того, каждый образ исполняемого файла имеет набор глобальных флагов, позволяющих включать код внутренней трассировки и контроля (хотя битовая структура этих флагов совершенно не соответствует структуре общесистемных глобальных флагов). Эти флаги не документированы, но могут пригодиться при изучении внутреннего устройства Windows.

К счастью, в Platform SDK и средствах отладки есть утилита Gflags.exe, позволяющая просматривать и изменять системные глобальные флаги (либо в реестре, либо в работающей системе) и глобальные флаги образов исполняемых файлов. Gflags поддерживает как GUI-интерфейс, так и командную строку. Параметры командной строки можно узнать, введя *gflags /?*. При запуске утилиты без параметров выводится диалоговое окно, показанное на рис. 3-28.

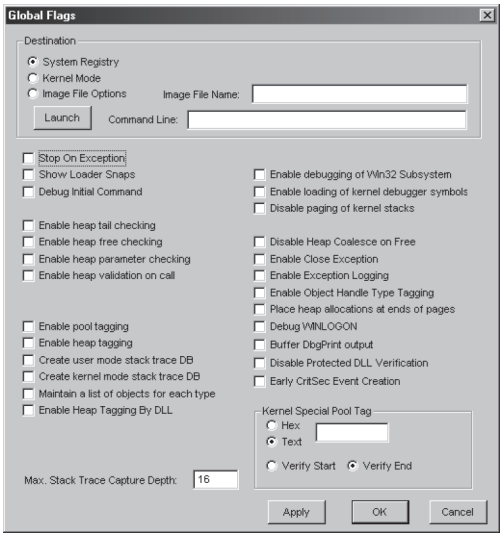


Рис. 3-28. *Настройка в Gflags системных отладочных параметров*

Вы можете переключаться между реестром (System Registry) и текущим значением переменной в системной памяти (Kernel Mode). Для внесения изменений нужно щелкнуть кнопку Apply (кнопка OK просто закрывает программу). Хотя вы можете изменять флаги в работающей системе, большинство из них требует перезагрузки для того, чтобы изменения вступили в силу.

Поскольку документации на этот счет нет, лучше перезагрузиться после любых изменений.

Выбрав Image File Options, вы должны ввести имя исполняемого в системе файла. Этот переключатель позволяет изменять набор глобальных флагов отдельного образа (а не всей системы). Заметьте, что флаги на рис. 3-29 отличаются от флагов на рис. 3-28.

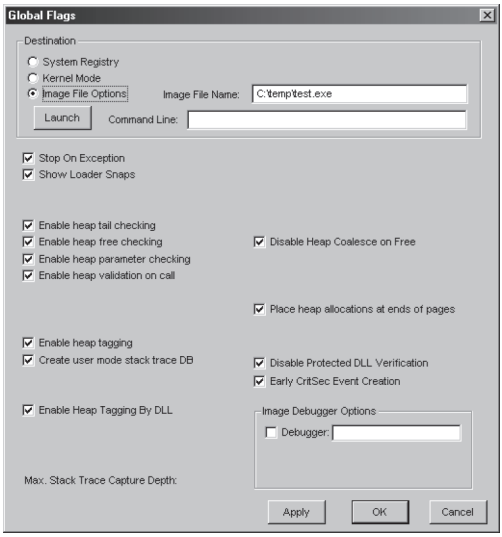


Рис. 3-29. *Настройка в Gflags глобальных флагов образа исполняемого файла*

ЭКСПЕРИМЕНТ: включение трассировки загрузчика образов и просмотр *NtGlobalFlag*

Чтобы увидеть пример детальной трассировочной информации, которую можно получить при установке глобальных флагов, попробуйте запустить Gflags в системе с загруженным отладчиком ядра, которая подключена к компьютеру с запущенной утилитой Kd или Windbg.

Далее попробуйте установить, например, глобальный флаг Show Loader Snaps. Для этого выберите Kernel Mode, установите флажок Show Loader Snaps и щелкните кнопку Apply. Теперь запустите на этой машине какую-нибудь программу, и отладчик ядра будет выдавать информацию, аналогичную показанной ниже.

```
LDR: PID: 0xb8 started - 'notepad'
LDR: NEW PROCESS
     Image Path: C:\Windows\system32\notepad.exe (notepad.exe)
     Current Directory: C:\ddk\bin
     Search Path: C:\Windows\System32;C:\Windows\system;C:\Windows
LDR: notepad.exe bound to comdlg32.dll
LDR: ntdll.dll used by comdlg32.dll
LDR: Snapping imports for comdlg32.dll from ntdll.dll
```

```
    ...
LDR: KERNEL32.dll loaded. - Calling init routine at 77f01000
LDR: RPCRT4.dll loaded. - Calling init routine at 77e1b6d5
LDR: ADVAPI32.dll loaded. - Calling init routine at 77dc1000
LDR: USER32.dll loaded. - Calling init routine at 77e78037
```

Для просмотра состояния переменной *NtGlobalFlag* можно использовать команды *!gflags* и *!gflag* отладчика ядра. Первая выводит список всех флагов, указывая, какие из них установлены, а *!gflag* показывает только установленные флаги.

```
kd> !gflags

NT!NtGlobalFlag 0x4400

  STOP_ON_EXCEPTION               SHOW_LDR_SNAPS
  DEBUG_INITIAL_COMMAND           STOP_ON_HUNG_GUI
  HEAP_ENABLE_TAIL_CHECK          HEAP_ENABLE_FREE_CHECK
  HEAP_VALIDATE_PARAMETERS        HEAP_VALIDATE_ALL
 *POOL_ENABLE_TAGGING             HEAP_ENABLE_TAGGING
  USER_STACK_TRACE_DB             KERNEL_STACK_TRACE_DB
 *MAINTAIN_OBJECT_TYPELIST        HEAP_ENABLE_TAG_BY_DLL
  ENABLE_CSRDEBUG                 ENABLE_KDEBUG_SYMBOL_LOAD
  DISABLE_PAGE_KERNEL_STACKS      HEAP_DISABLE_COALESCING
  ENABLE_CLOSE_EXCEPTIONS         ENABLE_EXCEPTION_LOGGING
  ENABLE_HANDLE_TYPE_TAGGING      HEAP_PAGE_ALLOCS
  DEBUG_INITIAL_COMMAND_EX        DISABLE_DBGPRINT

kd> !gflag
NtGlobalFlag at 8046a164
Current NtGlobalFlag contents: 0x00004400
    ptg - Enable pool tagging
    otl - Maintain a list of objects for each type
```

LPC

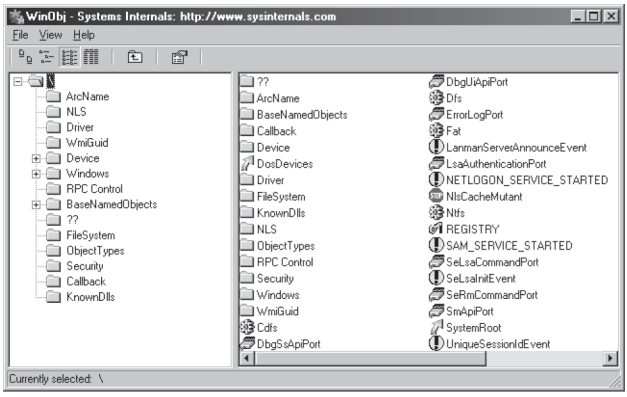
LPC (local procedure call) — это механизм межпроцессной связи для высокоскоростной передачи сообщений. Он недоступен через Windows API напрямую и является внутренним механизмом, которым пользуются только компоненты операционной системы Windows. Вот несколько примеров того, где применяется LPC.

- Windows-приложения, использующие RPC (документированный API), неявно используют и LPC, когда указывают локальный RPC — разновидность RPC, применяемую для взаимодействия между процессами в рамках одной системы.
- Некоторые функции Windows API обращаются к LPC, посылая сообщения процессу подсистемы Windows.

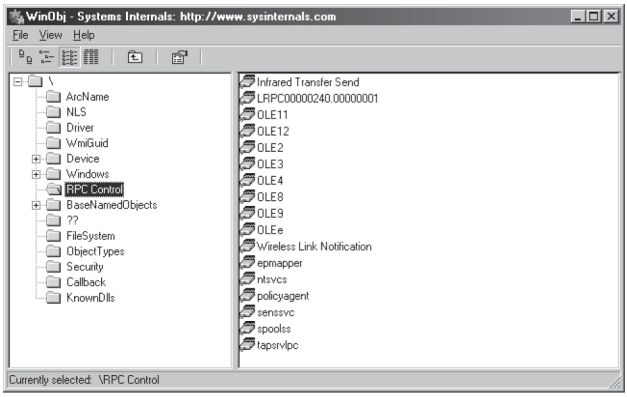
- Winlogon взаимодействует с процессом LSASS через LPC.
- Монитор состояния защиты (компонент исполнительной системы, рассматриваемый в главе 8) также взаимодействует с процессом LSASS через LPC.

ЭКСПЕРИМЕНТ: просмотр объектов «порт LPC»

Вы можете увидеть именованные объекты «порт LPC» (LPC port objects) с помощью утилиты Winobj с сайта *www.sysinternals.com*. Запустите Winobj.exe и выберите корневой каталог. Интересующие нас объекты обозначаются значком в виде разъема, как показано ниже.



Для просмотра объектов «порт LPC», используемых RPC, выберите каталог \RPC Control, как на следующей иллюстрации.



Вы также можете наблюдать объекты «порт LPC» с помощью команды *!lpc* отладчика ядра. Параметры этой команды позволяют перечислять порты LPC, сообщения LPC и потоки, ожидающие или посылающие эти сообщения. Для просмотра порта аутентификации LSASS (в него Winlogon посылает запросы на вход в систему) сначала нужно получить список портов в данной системе.

```
kd> !lpc
Usage:
    !lpc                      - Display this help
    !lpc message [MessageId] - Display the message with a
                               given ID and all related
                               information
                               If MessageId is not
                               specified, dump all messages
    !lpc port [PortAddress]  - Display the port information
    !lpc scan PortAddress    - Search this port and any
                               connected port
    !lpc thread [ThreadAddr] - Search the thread in rundown
                               port queues and display the
                               port info
                               If ThreadAddr is missing,
                               display all threads marked
                               as doing some lpc operations

kd> !lpc port
Scanning 206 objects
       1  Port: 0xe1360320 Connection: 0xe1360320
          Communication: 0x00000000 'SeRmCommandPort'
       1  Port: 0xe136bc20 Connection: 0xe136bc20
          Communication: 0x00000000 'SmApiPort'
       1  Port: 0xe133ba80 Connection: 0xe133ba80
          Communication: 0x00000000 'DbgSsApiPort'
       1  Port: 0xe13606e0 Connection: 0xe13606e0
          Communication: 0x00000000 'DbgUiApiPort'
       ...
       1  Port: 0xe205f040 Connection: 0xe205f040
          Communication: 0x00000000 'LsaAuthenticationPort'
       ...
```

Найдите порт с именем LsaAuthenticationPort и передайте команде *!lpc* его адрес, как показано ниже.

```
kd> !lpc port 0xe205f040
Server connection port e205f040 Name: LsaAuthenticationPort
    Handles: 1   References: 37
    Server process  : ff7d56c0 (lsass.exe)
    Queue semaphore : ff7bfcc8
    Semaphore state 0 (0x0)
    The message queue is empty
    The LpcDataInfoChainHead queue is empty
```

Как правило, LPC используются для взаимодействия между серверным процессом и одним или несколькими клиентскими процессами. LPC-соединение может быть установлено между двумя процессами пользовательского режима или между компонентом режима ядра и процессом пользователь-

ского режима. Например, как говорилось в главе 2, Windows-процессы иногда посылают сообщения подсистеме Windows через LPC. Некоторые системные процессы вроде Winlogon и LSASS тоже используют LPC. Примерами компонентов режима ядра, взаимодействующих с пользовательскими процессами через LPC, могут служить монитор состояния защиты и LSASS.

LPC предусматривает три способа обмена сообщениями.

- Сообщение длиной менее 256 байтов можно передать вызовом LPC с буфером, содержащим сообщение. Затем это сообщение копируется из адресного пространства процесса-отправителя в системное адресное пространство, а оттуда — в адресное пространство процесса-получателя.
- Если клиент и сервер хотят обменяться данными, размер которых превышает 256 байтов, они могут использовать общий раздел, на который они оба спроецированы. Отправитель помещает данные в общий раздел и посылает получателю уведомление с указателем на область раздела, где находятся данные.
- Если серверу нужно считать или записать данные, объем которых превышает размер общего раздела, то их можно напрямую считать из клиентского адресного пространства или записать туда. Для этого LPC предоставляет серверу две функции. Сообщение, посланное первой функцией, обеспечивает синхронизацию передачи последующих сообщений.

LPC экспортирует единственный объект исполнительной системы — объект «порт» (port object). Однако порты бывают нескольких видов.

- **Порт серверного соединения (server connection port)** Именованный порт, служащий точкой запроса связи с сервером. Через него клиенты могут соединяться с сервером.
- **Коммуникационный порт сервера (server communication port)** Безымянный порт, используемый сервером для связи с конкретным клиентом. У сервера имеется по одному такому порту на каждый активный клиент.
- **Коммуникационный порт клиента (client communication port)** Безымянный порт, используемый конкретным клиентским потоком для связи с конкретным сервером.
- **Безымянный коммуникационный порт (unnamed communication port)** Порт, создаваемый для связи между двумя потоками одного процесса.

LPC обычно используется так. Сервер создает именованный порт соединения. Клиент посылает в него запрос на установление связи. Если запрос удовлетворен, создается два безымянных порта — коммуникационный порт клиента и коммуникационный порт сервера. Клиент получает описатель коммуникационного порта клиента, а сервер — описатель коммуникационного порта сервера. После этого клиент и сервер используют новые порты для обмена данными.

Схема соединения между клиентом и сервером показана на рис. 3-30.

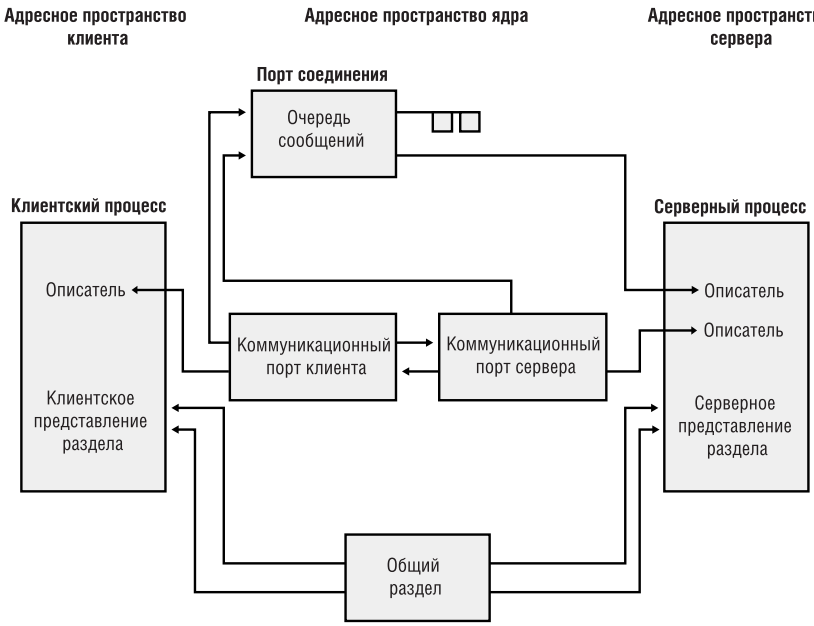


Рис. 3-30. *Использование LPC-портов*

Трассировка событий ядра

Различные компоненты ядра Windows и несколько базовых драйверов устройств оснащены средствами мониторинга для записи трассировочных данных об их работе, используемых при анализе проблем в системе. Эти компоненты опираются на общую инфраструктуру в ядре, которая предоставляет трассировочные данные механизму пользовательского режима — Event Tracing for Windows (ETW). Приложение, использующее ETW, попадает в одну или более следующих категорий.

- **Контроллер (controller)** Начинает и прекращает сеансы протоколирования (logging sessions), а также управляет буферными пулами.
- **Провайдер (provider)** Определяет GUID (globally unique identifiers) для классов событий, для которых он может создавать трассировочные данные, и регистрирует их в ETW. Провайдер принимает команды от контроллера на запуск и остановку трассировки классов событий, за которые он отвечает.
- **Потребитель (consumer)** Выбирает один или более сеансов трассировки, для которых ему нужно считывать трассировочные данные. Принимает информацию о событиях в буферы в режиме реального времени или в файлы журнала.

В системы Windows Server встроено несколько провайдеров пользовательского режима, в том числе для Active Directory, Kerberos и Netlogon. ETW

определяет сеанс протоколирования с именем NT Kernel Logger [также известный как регистратор ядра (kernel logger)] для использования ядром и базовыми драйверами. Провайдер для NT Kernel Logger реализуется драйвером устройства Windows Management Instrumentation (WMI) (драйвер называется Wmixwdm), который является частью Ntoskrnl.exe. (Подробнее о WMI см. соответствующий раздел в главе 5.) Этот драйвер не только служит основой регистратора ядра, но и управляет регистрацией классов событий ETW пользовательского режима.

Драйвер WMI экспортирует интерфейсы управления вводом-выводом для применения в ETW-процедурах пользовательского режима и драйверах устройств, предоставляющих трассировочные данные для регистратора ядра. (О командах управления вводом-выводом см. главу 9.) Он также реализует функции для использования компонентами в Ntoskrnl.exe режима ядра, которые формируют трассировочный вывод.

Когда в пользовательском режиме включается контроллер, регистратор ядра (библиотека ETW, реализованная в \Windows\System32\Ntdll.dll) посылает запрос управления вводом-выводом (I/O control request) драйверу WMI, сообщая ему, для каких классов событий контроллер хочет начать трассировку. Если настроено протоколирование в файлы журналов (в противоположность протоколированию в буфер памяти), драйвер WMI создает специальный системный поток в системном процессе, а тот создает файл журнала. Принимая события трассировки от активизированных источников трассировочных данных, драйвер WMI записывает их в буфер. Поток записи в журнал пробуждается раз в секунду, чтобы сбросить содержимое буферов в файл журнала.

Записи трассировки, генерируемые для регистратора ядра, имеют стандартный ETW-заголовок события трассировки, в котором содержатся временная метка, идентификаторы процесса и потока, а также сведения о том, какому классу события соответствует данная запись. Классы событий могут предоставлять дополнительные данные, специфичные для их событий. Например, класс дисковых событий (disk event class) указывает тип операции (чтение или запись), номер диска, на котором выполняется операция, а также смещение начального сектора и количество секторов, затрагиваемых данной операцией.

Классы трассировки, которые можно включить для регистратора ядра, и компонент, генерирующий каждый класс, перечислены ниже.

- **Дисковый ввод-вывод** Драйвер класса дисков.
- **Файловый ввод-вывод** Драйверы файловой системы.
- **Конфигурирование оборудования** Диспетчер Plug and Play (см. главу 9).
- **Загрузка/выгрузка образов** Системный загрузчик образов в ядре.
- **Ошибки страниц** Диспетчер памяти (см. главу 7).
- **Создание/удаление процессов** Диспетчер процессов (см. главу 6).
- **Создание/удаление потоков** Диспетчер процессов.

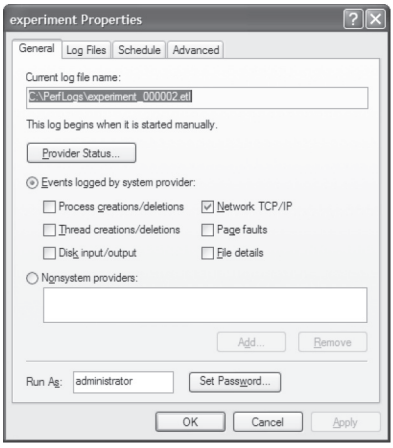
- **Операции с реестром** Диспетчер конфигурации (см. раздел «Реестр» в главе 4).
- **Активность TCP/UDP** Драйвер TCP/IP.

Более подробные сведения о ETW и регистраторе ядра, в том числе примеры кода для контроллеров и потребителей, см. в Platform SDK.

ЭКСПЕРИМЕНТ: трассировка активности TCP/IP с помощью регистратора ядра

Чтобы включить регистратор ядра и получить от него файл журнала активности TCP/IP, действуйте следующим образом.

1. Запустите оснастку Performance (Производительность) и выберите узел Performance Logs And Alerts (Журналы и оповещения производительности).
2. Укажите Trace Logs (Журналы трассировки) и выберите из меню Action (Действие) команду New Log Settings (Новые параметры журнала).
3. В появившемся окне присвойте имя новым параметрам (например, experiment).
4. В следующем диалоговом окне выберите Events Logged By System Provider (События, протоколируемые системным поставщиком) и сбросьте все, кроме Network TCP/IP (События сети TCP/IP).
5. В поле ввода Run As (От имени) введите имя учетной записи администратора и ее пароль.



6. Закройте это диалоговое окно и создайте активность в сети, открыв браузер и зайдя на какой-нибудь Web-сайт.
7. Укажите журнал трассировки, созданный в узле таких журналов, и выберите Stop (Остановка) из меню Action (Действие).
8. Откройте окно командной строки и перейдите в каталог C:\Perflogs (или тот каталог, который вы указали как место хранения файла журнала).

см. след. стр.

9. Если вы используете Windows XP или Windows Server 2003, запустите Tracerpt (эта утилита находится в каталоге \Windows\System32) и передайте ей имя файла журнала трассировки. Если вы работаете в Windows 2000, скачайте и запустите Tracedmp из ресурсов Windows 2000. Обе утилиты генерируют два файла: dumpfile.csv и summary.txt.
10. Откройте dumpfile.csv в Microsoft Excel или в любом текстовом редакторе. Вы должны увидеть записи трассировки TCP и/или UDP:

```
TcpIp Recv 0xFFFFFFFF 1.27E+17 0 0 4 88 192.168.001.101 192.168.001.108
4608 0 0 0
TcpIp Send 0xFFFFFFFF 1.27E+17 0 0 4 76 192.168.001.101 192.168.001.108
4608 0 0 0
TcpIp Recv 0xFFFFFFFF 1.27E+17 0 0 4 88 192.168.001.101 192.168.001.108
4608 0 0 0
TcpIp Send 0xFFFFFFFF 1.27E+17 0 0 4 76 192.168.001.101 192.168.001.108
4608 0 0 0
TcpIp Recv 0xFFFFFFFF 1.27E+17 0 0 4 88 192.168.001.101 192.168.001.108
4608 0 0 0
TcpIp Send 0xFFFFFFFF 1.27E+17 0 0 4 76 192.168.001.101 192.168.001.108
4608 0 0 0
TcpIp Recv 0xFFFFFFFF 1.27E+17 0 0 4 88 192.168.001.101 192.168.001.108
4608 0 0 0
TcpIp Send 0xFFFFFFFF 1.27E+17 0 0 4 76 192.168.001.101 192.168.001.108
4608 0 0 0
```

Wow64

Wow64 (эмуляция Win32 в 64-разрядной Windows) относится к программному обеспечению, которое дает возможность выполнять 32-разрядные x86-приложения в 64-разрядной Windows. Этот компонент реализован как набор DLL пользовательского режима.

- Wow64.dll — управляет созданием процессов и потоков, подключается к диспетчеризации исключений и перехватывает вызовы базовых системных функций, экспортируемых Ntoskrnl.exe. Также реализует перенаправление файловой системы (file system redirection) и перенаправление реестра и отражение (reflection).
- Wow64Cpu.dll — управляет 32-разрядным контекстом процессора каждого потока, выполняемого внутри Wow64, и предоставляет специфичную для процессорной архитектуры поддержку переключения режима процессора из 32-разрядного в 64-разрядный и наоборот.
- Wow64Win.dll — перехватывает вызовы системных GUI-функций, экспортируемых Win32k.sys.

  Взаимосвязь этих DLL показана на рис. 3-31.

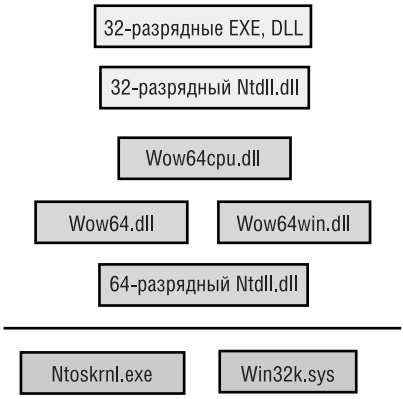


Рис. 3-31. *Архитектура Wow64*

Системные вызовы

Wow64 ставит ловушки на всех путях выполнения, где 32-разрядный код должен взаимодействовать с родным 64-разрядным или где 64-разрядной системе нужно обращаться к 32-разрядному коду пользовательского режима. При создании процесса диспетчер процессов проецирует на его адресное пространство 64-разрядную библиотеку Ntdll.dll. Загрузчик 64-разрядной системы проверяет заголовок образа и, если этот процесс 32-разрядный для платформы x86, загружает Wow64.dll. После этого Wow64 проецирует 32-разрядную Ntdll.dll (она хранится в каталоге \Windows\Syswow64). Далее Wow64 настраивает стартовый контекст внутри Ntdll, переключает процессор в 32-разрядный режим и начинает выполнять 32-разрядный загрузчик. С этого момента все идет так же, как в обычной 32-разрядной системе.

Специальные 32-разрядные версии Ntdll.dll, User32.dll и Gdi32.dll находятся в каталоге \Windows\Syswow64. Они вызывают Wow64, не выдавая инструкции вызова, которые используются в истинно 32-разрядной системе. Wow64 переключается в «родной» 64-разрядный режим, захватывает параметры, связанные с системным вызовом, преобразует 32-разрядные указатели в 64-разрядные и выдает соответствующий для 64-разрядной системы системный вызов. Когда последняя возвращает управление, Wow64 при необходимости преобразует любые выходные параметры из 64-битных в 32-битные форматы и вновь переключается в 32-разрядный режим.

Диспетчеризация исключений

Wow64 перехватывает диспетчеризацию исключений через *KiUserExceptionDispatcher* в Ntdll. Всякий раз, когда 64-разрядное ядро собирается направить исключение Wow64-процессу, Wow64 перехватывает его и запись контекста (context record) в пользовательском режиме, а затем, создав на их основе 32-разрядные исключение и запись контекста, направляет их своему процессу так же, как это сделало бы истинно 32-разрядное ядро.

Пользовательские обратные вызовы

Wow64 перехватывает все обратные вызовы из режима ядра в пользовательский режим. Wow64 интерпретирует их как системные вызовы; однако трансляция данных происходит в обратном порядке: входные параметры преобразуются из 64-битных форматов в 32-битные, а выходные (после возврата из обратного вызова) — из 32-битных в 64-битные.

Перенаправление файловой системы

Чтобы обеспечить совместимость приложений и упростить перенос Win32-программ на платформу 64-разрядной Windows, имена системных каталогов сохранены прежними. Поэтому в \Windows\System32 содержатся «родные» 64-разрядные исполняемые файлы. Так как Wow64 ставит ловушки на все системные вызовы, этот компонент транслирует все API-вызовы, относящиеся к путям, и заменяет в них каталог \Windows\System32 на \Windows\Syswow64. Wow64 также перенаправляет \Windows\System32 \Ime в \Windows\System32\IME (x86), чтобы обеспечить совместимость 32-разрядных приложений в 64-разрядных системах с установленной поддержкой дальневосточных языков. Кроме того, 32-разрядные программы устанавливаются в каталог \Program Files (x86), тогда как 64-разрядные — в обычный каталог \Program Files.

В каталоге \Windows\System32 есть несколько подкаталогов, которые по соображениям совместимости исключаются из перенаправления. Так что, если 32-разрядным приложениям понадобится доступ к этим каталогам, они смогут обращаться к ним напрямую. В число таких каталогов входят:

- %windir%\system32\drivers\etc;
- %windir%\system32\spool;
- %windir%\system32\catroot2;
- %windir%\system32\logfiles.

Наконец, Wow64 предоставляет механизм, позволяющий отключать перенаправление файловой системы, встроенное в Wow64, для каждого потока индивидуально. Данный механизм доступен через функцию *Wow64EnableWow64FsRedirection*, которая впервые появилась в Windows Server 2003.

Перенаправление реестра и отражение

Приложения и компоненты хранят свои конфигурационные данные в реестре. Эту информацию компоненты обычно записывают в реестр при регистрации в ходе установки. Если один и тот же компонент поочередно устанавливается и регистрируется как 32- и 64-разрядный, тогда компонент, зарегистрированный последним, переопределяет регистрацию предыдущего, поскольку оба они пишут по одному адресу в реестре.

Чтобы решить эту проблему, не модифицируя 32-разрядные компоненты, реестр делится на две части: Native и Wow64. По умолчанию 32-разрядные

компоненты получают доступ к 32-разрядному представлению реестра, а 64-разрядные — к 64-разрядному представлению. Это создает безопасную среду исполнения для 32- и 64-разрядных компонентов и отделяет состояние 32-разрядных приложений от состояния 64-разрядных (если таковые есть).

Реализуя это решение, Wow64 перехватывает все системные вызовы, открывающие разделы реестра, и модифицирует пути к разделам так, чтобы они указывали на контролируемое Wow64 представление реестра. Wow64 разбивает реестр в следующих точках:

- HKLM\Software;
- HKEY_CLASSES_ROOT;
- HKEY_CURRENT_USER\Software\Classes.

В каждом из этих разделов Wow64 создает раздел с именем Wow6432-Node. В нем сохраняется конфигурационная информация 32-разрядного программного обеспечения. Остальные части реестра 32- и 64-разрядные приложения используют совместно (например, HKLM\System).

При вызове функций *RegOpenKeyEx* и *RegCreateKeyEx* приложения могут передавать следующие флаги:

- KEY_WOW64_64KEY — для явного открытия 64-разрядного раздела из 32- или 64-разрядного приложения;
- KEY_WOW64_32KEY — для явного открытия 32-разрядного раздела из 32- или 64-разрядного приложения.

Для обеспечения взаимодействия через 32- и 64-разрядные COM-компоненты Wow64 отражает изменения в некоторых частях одного представления реестра на другое. Для этого Wow64 перехватывает операции обновления любого из отслеживаемых разделов в одном из представлений и отражает соответствующие изменения на другое представление. Вот список отслеживаемых разделов:

- HKLM\Software\Classes;
- HKLM\Software\Ole;
- HKLM\Software\Rpc;
- HKLM\Software\COM3;
- HKLM\Software\EventSystem.

Wow64 использует интеллектуальный подход к отражению HKLM\Software\Classes\CLSID: транслируются только CLSID-идентификаторы LocalServer32, так как они могут быть COM-активированы 32- или 64-разрядными приложениями, а CLSID-идентификаторы InProcServer32 не отражаются, поскольку 32-разрядные COM DLL нельзя загрузить в 64-разрядный процесс, равно как и 64-разрядные COM DLL в 32-разрядный процесс.

При отражении раздела или параметра механизм отражения реестра (registry reflector) помечает раздел так, чтобы было понятно, что он создан именно этим механизмом. Это позволяет ему выбирать дальнейший алгоритм действий при удалении отражаемого раздела.

Запросы управления вводом-выводом

Приложения могут не только выполнять обычные операции чтения и записи, но и взаимодействовать с некоторыми драйверами устройств через интерфейс управления вводом-выводом на устройствах, используя API-функцию *DeviceIoControlFile*. При ее вызове можно указать входной и/или выходной буфер. Если он содержит данные, зависимые от указателя, и процесс, посылающий запрос, является Wow64-процессом, тогда у 32-разрядного приложения и 64-разрядного драйвера разные представления входной и/или выходной структуры, так как 32-разрядные программы используют указатели длиной 4 байта, а 64-разрядные — длиной 8 байтов. В этом случае предполагается, что драйвер режима ядра сам преобразует соответствующие структуры, зависимые от указателей. Чтобы определить, исходит ли запрос от Wow64-процесса, драйверы могут вызывать функцию *IoIs32bitProcess*.

16-разрядные программы установки

Wow64 не поддерживает выполнение 16-разрядных приложений. Но поскольку многие программы установки являются 16-разрядными, в Wow64 предусмотрен специальный код, все же позволяющий выполнять 16-разрядные программы установки общеизвестных приложений. К таким средствам установки, в частности, относятся:

- Microsoft ACME Setup версий 2.6, 3.0, 3.01 и 3.1;
- InstallShield версий 5.*x*.

Всякий раз, когда с помощью API-функции *CreateProcess* предпринимается попытка создать 16-разрядный процесс, система загружает Ntvdm64.dll и передает ей управление, чтобы та определила, относится ли данный 16-разрядный исполняемый файл к одной из поддерживаемых программ установки. Если да, то выдается другой вызов *CreateProcess*, чтобы запустить 32-разрядную версию этого установщика с теми же аргументами командной строки.

Печать

Использовать 32-разрядные драйверы принтера в 64-разрядной Windows нельзя. Они должны быть 64-разрядными версиями, «родными» для данной системы. Однако, поскольку драйверы принтера работают в пользовательском адресном пространстве запрашивающего процесса, а 64-разрядная Windows поддерживает лишь истинно 64-разрядные драйверы принтера, нужен специальный механизм для поддержки печати из 32-разрядных процессов. Для этого все вызовы функций печати перенаправляются в Splwow64.exe — RPC-сервер печати Wow64. Так как Splwow64 является 64-разрядным процессом, он может загрузить 64-разрядные драйверы принтера.

Ограничения

Wow64 (в отличие от 32-разрядных версий Windows) не поддерживает выполнение 16-разрядных приложений или загрузку 32-разрядных драйверов

устройств режима ядра (их нужно перевести в истинно 64-разрядные). Wow64-процессы могут загружать лишь 32-разрядные DLL (загрузка истинно 64-разрядных DLL невозможна). Аналогичным образом 64-разрядные процессы не могут загружать 32-разрядные DLL.

В дополнение к сказанному Wow64 в системах IA64 из-за различий в размерах страниц памяти не поддерживает функции *ReadFileScatter*, *WriteFileGather*, *GetWriteWatch* или Address Window Extension (AWE). Кроме того, Wow64-процессам недоступно аппаратное ускорение операций через DirectX (таким процессам предоставляется лишь программная эмуляция).

Резюме

В этой главе мы изучили важнейшие базовые механизмы, на которых построена исполнительная система Windows. В следующей главе будут рассмотрены три важных механизма, образующих инфраструктуру управления в Windows: реестр, сервисы и WMI (Windows Management Instrumentation).

ГЛАВА 4

Механизмы управления

В этой главе описываются три фундаментальных механизма Microsoft Windows, критически важных для управления системой и ее конфигурирования:

- реестр;
- сервисы;
- Windows Management Instrumentation (Инструментарий управления Windows).

Реестр

Реестр играет ключевую роль в конфигурировании и управлении Windows. Это хранилище общесистемных и пользовательских параметров. Реестр не является статичной совокупностью хранящихся на жестком диске данных, как думают многие. Прочитав этот раздел, вы увидите, что он представляет собой окно в мир различных структур, которые хранятся в памяти компьютера и поддерживаются ядром и исполнительной системой. Данный раздел не претендует на роль полного справочника по реестру Windows. Исчерпывающая информация такого рода для Windows 2000 находится в справочном файле «Technical Reference to the Windows 2000 Registry» (Regentry.chm), который поставляется с ресурсами Windows 2000, а для Windows XP и Windows Server 2003 эта информация доступна через Интернет по ссылке *http://www.microsoft.com/windowsserver2003/techinfo/reskit/deploykit.mspx*.

Мы начнем с обзора структуры реестра, рассмотрим поддерживаемые им типы данных и ключевую информацию, хранящуюся в реестре Windows. Потом заглянем поглубже внутрь и обсудим механизмы, используемые диспетчером конфигурации — компонентом исполнительной системы, который отвечает за реализацию базы данных реестра. Среди прочего мы коснемся внутренней структуры реестра на диске, способов выборки конфигурационной информации по запросу приложений и мер защиты этой важнейшей системной базы данных.

Просмотр и изменение реестра

Как правило, следует избегать прямого редактирования реестра — приложения и система, хранящие в реестре параметры, которые могут потребовать настройки вручную, должны предоставлять соответствующий пользователь-

ский интерфейс (UI) для их модификации. Однако, как вы уже неоднократно видели в этой книге, для изменения некоторых дополнительных и отладочных параметров никакого UI не предусмотрено. Поэтому в Windows включен ряд утилит, позволяющих просматривать и модифицировать реестр.

Windows 2000 поставляется с двумя утилитами для редактирования реестра — Regedit.exe и Regedt32.exe, — тогда как в Windows XP и Windows Server 2003 имеется лишь Regedit.exe. Причина в том, что версия Regedit в Windows 2000 была перенесена из Windows 98 и поэтому не поддерживала редактирование или просмотр параметров защиты и типов данных, не определенных в Windows 98. Поэтому в Windows 2000 была добавлена Regedt32, которая не обладала развитыми средствами поиска и поддержки импорта/экспорта, но поддерживала параметры защиты и специфичные для Windows 2000 типы данных. Regedit, поставляемая с Windows XP и Windows Server 2003, распознает все типы данных в реестре и позволяет редактировать параметры защиты, ввиду чего необходимость в Regedt32 отпала.

Существует также целый ряд утилит для работы с реестром из командной строки. Например, Reg.exe, включенная в Windows XP и Windows Server 2003 и доступная в Windows 2000 Support Tools, дает возможность импортировать, экспортировать, создавать резервные копии и восстанавливать разделы реестра, а также сравнивать, модифицировать и удалять разделы и параметры.

Использование реестра

Конфигурационные данные всегда считываются в следующих случаях.

- В ходе загрузки система читает параметры, указывающие, какие драйверы устройств нужно загрузить, а различные подсистемы (вроде диспетчера памяти и диспетчера процессов) — параметры, позволяющие им настраивать себя и поведение системы.
- При входе Explorer и другие Windows-компоненты считывают из реестра предпочтения данного пользователя, в том числе буквы подключенных сетевых дисков, размещение ярлыков, а также настройки рабочего стола, меню и др.
- При запуске приложения считывают общесистемные параметры, например список дополнительных установленных компонентов, информацию о лицензировании, настройки для данного пользователя (меню, размещение панелей инструментов, список недавно открывавшихся документов и т. д.).

Однако чтение реестра возможно и в другие моменты, скажем, в ответ на модификацию его параметра или раздела. Некоторые приложения ведут мониторинг своих конфигурационных параметров в реестре и считывают обновленные значения, как только обнаруживают изменения. Но в целом, если система простаивает, работы с реестром не должно быть.

Реестр обычно модифицируется в следующих ситуациях.

- Исходная структура реестра и многие настройки по умолчанию определяются его прототипной версией, поставляемой на дистрибутиве Windows и копируемой при установке новой системы.

- Программы установки различных приложений создают для них настройки по умолчанию и настройки, отражающие выбор пользователя в процессе установки.
- При установке драйвера устройства подсистема Plug and Play создает разделы и параметры в реестре, которые сообщают диспетчеру ввода-вывода, как запускать драйвер, а также создает другие параметры, определяющие работу этого драйвера. (Подробнее об установке драйверов устройств см. главу 9.)
- Когда вы изменяете параметры приложения или системы через UI, эти изменения часто сохраняются в реестре.

ПРИМЕЧАНИЕ Как ни печально, но некоторые приложения периодически опрашивают реестр на предмет изменений, тогда как делать это следует через функцию *RegNotifyChangeKey*, которая отправляет поток в сон до тех пор, пока в интересующей его части реестра не произойдет какое-нибудь изменение.

Типы данных в реестре

Реестр — это база данных, структура которой аналогична структуре логического тома. Он содержит *разделы* (keys), напоминающие дисковые каталоги, и *параметры* (values), которые можно сравнить с файлами на диске. Раздел представляет собой контейнер, содержащий другие разделы, называемые *подразделами* (subkeys), и/или параметры. Параметры хранят собственно данные. Разделы верхнего уровня называются *корневыми*. Здесь мы будем использовать термины «подраздел» и «раздел» как синонимы (лишь корневые разделы не могут быть подразделами).

Соглашение об именовании разделов и параметров заимствовано из файловой системы. Таким образом, параметру можно присвоить имя, которое сохраняется в каком-либо разделе. Исключением из этой схемы является безымянный параметр, присутствующий в каждом разделе. Утилиты реестра, Regedit и Regedt32, по-разному показывают этот параметр: Regedit обозначает его как (Default) [(По умолчанию)], а Regedt32 — как <No Name> (<БЕЗ ИМЕНИ>).

В параметрах хранятся данные 15 типов, перечисленных в таблице 4-1. Большинство параметров реестра имеет тип REG_DWORD, REG_BINARY или REG_SZ. Параметры типа REG_DWORD содержат числовые или булевы значения, параметры типа REG_BINARY — данные, требующие более 32 битов, или произвольные двоичные данные (например зашифрованные пароли), а параметры типа REG_SZ — строки (естественно, в Unicode-формате), которые могут представлять такие элементы, как имена, пути, типы и имена файлов.

Таблица 4-1. *Типы параметров реестра*

Тип параметра	Описание
REG_NONE	Нетипизированный параметр
REG_SZ	Unicode-строка фиксированной длины
REG_EXPAND_SZ	Unicode-строка переменной длины; может включать переменные окружения
REG_BINARY	Двоичные данные произвольной длины
REG_DWORD	32-битное число
REG_DWORD_LITTLE_ENDIAN	32-битное число, в котором первым является младший байт; эквивалентно REG_DWORD
REG_DWORD_BIG_ENDIAN	32-битное число, в котором первым является старший байт
REG_LINK	Символьная ссылка в формате Unicode
REG_MULTI_SZ	Массив Unicode-строк с завершающим нулем
REG_RESOURCE_LIST	Описание аппаратного ресурса
REG_FULL_RESOURCE_DESCRIPTOR	Описание аппаратного ресурса
REG_RESOURCE_REQUIREMENTS_LIST	Список требований к ресурсам
REG_QWORD	64-битное число
REG_QWORD_LITTLE_ENDIAN	64-битное число, в котором первым является младший байт; эквивалентно REG_QWORD
REG_QWORD_BIG_ENDIAN	64-битное число, в котором первым является старший байт

Особенно интересен тип REG_LINK, поскольку он позволяет разделу ссылаться на другой раздел или параметр. Например, если параметр \Root1\Link содержит значение \Root2\RegKey типа REG_LINK, а параметр RegKey — RegValue, то значение RegValue можно идентифицировать двумя путями: \Root1\Link\RegValue и \Root2\RegKey\RegValue. Как поясняется в следующем разделе, Windows интенсивно использует ссылки в реестре: три из шести корневых разделов реестра представляют собой ссылки на подразделы трех корневых разделов, которые ссылками не являются. Ссылки не записываются на диск, а создаются динамически при каждой загрузке системы.

Логическая структура реестра

Вы можете проследить схему организации реестра через данные, которые в нем хранятся. Существует шесть корневых разделов (добавлять или удалять корневые разделы нельзя), описанных в таблице 4-2.

Таблица 4-2. *Корневые разделы реестра*

Корневой раздел	Описание
HKEY_CURRENT_USER	Содержит данные, сопоставленные с пользователем, который вошел в систему на данный момент
HKEY_USERS	Хранит информацию обо всех учетных записях на компьютере
HKEY_CLASSES_ROOT	Хранит сопоставления файлов и регистрационную информацию COM-объектов
HKEY_LOCAL_MACHINE	Содержит информацию, специфичную для системы
HKEY_PERFORMANCE_DATA	Хранит сведения о производительности
HKEY_CURRENT_CONFIG	Включает некоторые сведения о текущем профиле оборудования

Почему имена корневых разделов начинаются с буквы «H»? Дело в том, что имена корневых разделов представляют Windows-описатели (**H**andles) разделов (**KEY**). Как говорилось в главе 1, HKLM является аббревиатурой HKEY_LOCAL_MACHINE. В таблице 4-3 приводится список всех корневых разделов и их аббревиатур. Содержимое и предназначение каждого из них подробно обсуждаются в следующих разделах главы.

Таблица 4-3. *Предназначение корневых разделов реестра*

Корневой раздел	Аббревиатура	Описание	Ссылка
HKEY_CURRENT_USER	HKCU	Ссылается на профиль пользователя, вошедшего в систему	На подраздел в HKEY_USERS, соответствующий текущему вошедшему в систему пользователю
HKEY_USERS	HKU	Содержит подразделы для всех загруженных профилей пользователей	Не является ссылкой
HKEY_CLASSES_ROOT	HKCR	Содержит сведения о сопоставлениях файлов и регистрационную информацию COM	HKLM\SOFTWARE\Classes
HKEY_LOCAL_MACHINE	HKLM	Контейнер, содержащий другие разделы	Не является ссылкой
HKEY_CURRENT_CONFIG	HKCC	Текущий профиль оборудования	HKLM\SYSTEM\CurrentControlSet\HardwareProfiles\Current
HKEY_PERFORMANCE_DATA	HKPD	Счетчики производительности	Не является ссылкой

HKEY_CURRENT_USER

Корневой раздел HKCU содержит данные о предпочтениях и конфигурации программного обеспечения для локально зарегистрированного пользовате-

ля. Этот раздел ссылается на профиль текущего пользователя, находящийся на жестком диске в файле \Documents and Settings\<*имя_пользователя*>\Ntuser.dat (описание файлов реестра см. в разделе «Внутренние механизмы реестра» далее в этой главе). При каждой загрузке профиля пользователя (например, при регистрации в системе или при выполнении сервисного процесса в увязке с именем какого-либо пользователя) HKCU создается как ссылка на подраздел соответствующего пользователя в HKEY_USERS. Некоторые подразделы HKCU перечислены в таблице 4-4.

Таблица 4-4. *Подразделы в HKEY_CURRENT_USER*

Подраздел	Описание
AppEvents	Сопоставления звуковых сигналов с событиями
Console	Параметры окна командной строки (ширина, высота, цвет и т. д.)
Control Panel	Текущая экранная заставка, оформление рабочего стола, параметры клавиатуры и мыши, настройки специальных возможностей, а также язык и региональные стандарты
Environment	Определения переменных окружения
Keyboard Layout	Раскладки клавиатуры
Network	Имена и параметры подключенных сетевых дисков
Printers	Параметры подключения принтеров
Software	Настройки программ, специфичные для пользователя
Программные группы	Группы главного меню, специфичные для пользователя
Информация о переходе с Windows 3.1	Данные о состоянии файлов для систем, обновляемых с Windows 3.x до Windows 2000 и выше

HKEY_USERS

HKU содержит подраздел для каждого загруженного профиля пользователя, регистрационную базу данных классов и подраздел HKU\.DEFAULT, связанный с профилем для системы (этот профиль предназначен для процессов, выполняемых под локальной системной учетной записью; см. раздел «Сервисы» далее в этой главе). Данный профиль используется Winlogon, например, чтобы изменения в параметрах фона рабочего стола были реализованы на экране входа. Если пользователь входит в систему в первый раз и если его учетная запись не зависит от доменного профиля роуминга (т. е. профиль пользователя извлекается из централизованного хранилища в сети по указанию контроллера домена), система создает профиль для его учетной записи на основе профиля, хранящегося в каталоге C:\Documents and Settings\Default User.

Каталог, где система хранит профили, определяется параметром реестра HKLM\Software\Microsoft\Windows NT\CurrentVersion\ProfileList\ProfilesDirectory, который по умолчанию устанавливается в %SystemDrive%\Documents and Settings. Раздел ProfileList также хранит список профилей, имеющихся в системе. Информация по каждому профилю помещается в подраздел, имя

которого отражает SID учетной записи, соответствующей данному профилю (сведения о SID см. в главе 8). Информация в разделе профиля включает время последней загрузки этого профиля (параметры *ProfileLoadTimeLow* и *ProfileLoadTimeHigh*), двоичное представление SID учетной записи (параметр *Sid*) и путь к кусту профиля на диске в каталоге ProfileImagePath (о кустах см. раздел «Кусты» далее в этой главе). Windows XP и Windows Server 2003 показывают список профилей в диалоговом окне управления профилями пользователей, которое представлено на рис. 4-1. Чтобы открыть это окно, запустите апплет System (Система) из Control Panel (Панель управления), перейдите на вкладку Advanced (Дополнительно) и в разделе User Profiles (Профили пользователей) щелкните кнопку Settings (Параметры).

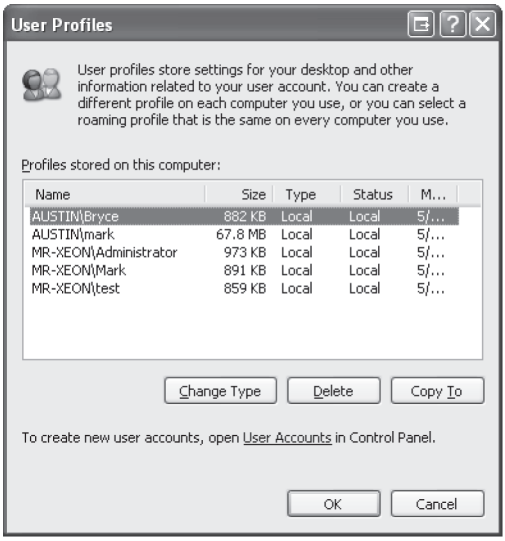


Рис. 4-1. *Диалоговое окно User Profiles (Профили пользователей)*

ЭКСПЕРИМЕНТ: наблюдение за загрузкой и выгрузкой профилей

Чтобы увидеть, как профиль загружается в реестр, а потом выгружается, запустите командой *runas* какой-нибудь процесс под учетной записью пользователя, не вошедшего на данный момент в систему. Пока новый процесс выполняется, запустите Regedit и обратите внимание на загруженный раздел профиля в HKEY_USERS. После завершения процесса нажмите в Regedit клавишу F5 для обновления, и этого профиля в реестре больше не будет.

HKEY_CLASSES_ROOT

HKCR включает информацию двух типов: сопоставления расширений файлов и регистрационные данные COM-классов. Для каждого зарегистрированного типа файлов существует свой раздел. Большинство разделов содержит

параметры типа REG_SZ, ссылающиеся на другие разделы HKCR, где находится информация о сопоставлениях классов файлов. Например, HKCR\.xls ссылается на сведения о файлах Microsoft Excel в разделе HKCU\Excel.Sheet.8 (последняя цифра указывает на версию Microsoft Excel). Другие разделы содержат детальную информацию о конфигурации COM-объектов, зарегистрированных в системе.

Раздел HKEY_CLASSES_ROOT формируется на основе:

- специфичных для конкретного пользователя регистрационных данных классов в HKCU\SOFTWARE\Classes (хранятся в \Documents and Settings\ *<имя_пользователя>*\Local Settings\Application Data\Microsoft\Windows\ Usrclass.dat);
- общесистемных регистрационных данных классов в HKLM\SOFTWARE\ Classes.

Причина, по которой регистрационные данные, специфичные для каждого пользователя, были отделены от общесистемных, заключается в том, что это дает возможность включать соответствующие настройки и в профили «блуждающих» пользователей (профили роуминга). Это же устранило дыру в защите: непривилегированный пользователь не может изменить или удалить разделы в HKEY_CLASSES_ROOT и тем самым повлиять на функционирование приложений в системе. Непривилегированные пользователи и приложения могут считывать общесистемные данные и добавлять новые разделы и параметры в общесистемные данные (которые отражаются на данные, специфичные для этих пользователей), но изменять существующие разделы и параметры им разрешается лишь в собственных данных.

HKEY_LOCAL_MACHINE

HKLM — корневой раздел, содержащий подразделы с общесистемной конфигурационной информацией: HARDWARE, SAM, SECURITY, SOFTWARE и SYSTEM.

Подраздел HKLM\HARDWARE содержит описание аппаратного обеспечения системы и все сопоставления драйверов с устройствами. Диспетчер устройств, который запускается с вкладки Hardware (Оборудование) окна свойств системы, позволяет просматривать информацию об устройствах, получаемую простым считыванием значений параметров из раздела HARDWARE.

ЭКСПЕРИМЕНТ: забавы с разделом Hardware

Вы можете обмануть своих коллег или друзей, заставив их поверить в то, что у вас самый последний процессор, модифицировав параметр *ProcessorNameString* в разделе HKLM\HARDWARE\DESCRIPTION\System\ CentralProcessor\0. Апплет System (Система) отображает значение параметра *ProcessorNameString* на вкладке General (Общие). Но изменение остальных параметров никак не влияет на информацию, выводимую апплетом System, так как система кэширует многие параметры для использования функциями, через которые приложения запрашивают у системы возможности установленного на данном компьютере процессора.

В HKLM\SAM находится информация о локальных учетных записях и группах, например пароли, определения групп и сопоставления с доменами. Система Windows Server, работающая как контроллер домена, хранит доменные и групповые учетные записи в Active Directory — базе данных, которая содержит общедоменные параметры и сведения. (Active Directory в этой книге не рассматривается.) По умолчанию дескриптор защиты раздела SAM сконфигурирован так, что к нему не имеет доступа даже администратор.

В HKLM\SECURITY хранятся данные, которые относятся к общесистемным политикам безопасности, а также сведения о правах, назначенных пользователям. HKLM\SAM связан с подразделом SECURITY в разделе HKLM\SECURITY\SAM. По умолчанию содержимое HKLM\SECURITY недоступно для просмотра, поскольку параметры защиты разрешают доступ только по учетной записи System. Вы можете сменить дескриптор защиты, чтобы администраторы получили доступ к этому разделу для чтения, или, если вам любопытно, что там находится, запустить Regedit под локальной системной учетной записью с помощью PsExec (как это сделать, будет показано в соответствующем эксперименте). Но это почти ничего не даст, так как данные в нем не документированы, а пароли зашифрованы (по алгоритму необратимого шифрования).

HKLM\SOFTWARE — то место, где Windows хранит общесистемную конфигурационную информацию, не требуемую при загрузке системы. Кроме того, здесь сохраняют свои общесистемные настройки приложения сторонних разработчиков (пути к файлам, каталоги приложений, даты лицензий и сроки их окончания).

HKLM\SYSTEM содержит общесистемную конфигурационную информацию, необходимую для загрузки системы, например списки загружаемых драйверов и запускаемых сервисов. Поскольку эта информация критична для запуска системы, Windows делает ее копию, называемую *последней удачной конфигурацией* (last known good control set). Она позволяет вернуться к последней работоспособной конфигурации, если после изменений, внесенных в текущую конфигурацию, система перестала загружаться. Подробнее об этом — ближе к концу главы.

HKEY_CURRENT_CONFIG

HKEY_CURRENT_CONFIG — просто ссылка на текущий профиль оборудования, хранящийся в HKLM\SYSTEM\CurrentControlSet\Hardware Profiles\Current. Профили оборудования позволяют администратору изменять базовые настройки системных драйверов. Хотя реальный профиль может меняться от загрузки к загрузке, благодаря разделу HKCC приложения всегда имеют дело с текущим активным профилем. Управление профилями оборудования осуществляется через диалоговое окно Hardware Profiles (Профили оборудование), которое открывается кнопкой Settings (Профили оборудования) в одноименном разделе на вкладке Hardware (Оборудование) в апплете System. При загрузке Ntldr предложит указать, какой профиль вам нужен, если он не один.

HKEY_PERFORMANCE_DATA

Реестр также является механизмом, который в Windows обеспечивает доступ к значениям счетчиков производительности. При этом не важно, предоставлены счетчики компонентами операционной системы или серверными приложениями. Одна из дополнительных выгод обращения к счетчикам производительности через реестр — возможность удаленного мониторинга рабочих характеристик без лишних издержек, поскольку удаленный доступ к реестру легко получить через обычные API-функции реестра.

Обратиться напрямую к этим данным можно только программным путем через Windows-функции реестра типа *RegQueryValueEx*, открыв специальный раздел с именем HKEY_PERFORMANCE_DATA. Доступ к разделу HKPD из редактора реестра невозможен — здесь хранится не сама информация о производительности, а ссылки на соответствующие источники этих данных.

Информация, относящаяся к счетчикам производительности, доступна и через функции Performance Data Helper (PDH), предоставляемые Performance Data Helper API (Pdh.dll). Компоненты, используемые для получения значений счетчиков производительности, показаны на рис. 4-2.

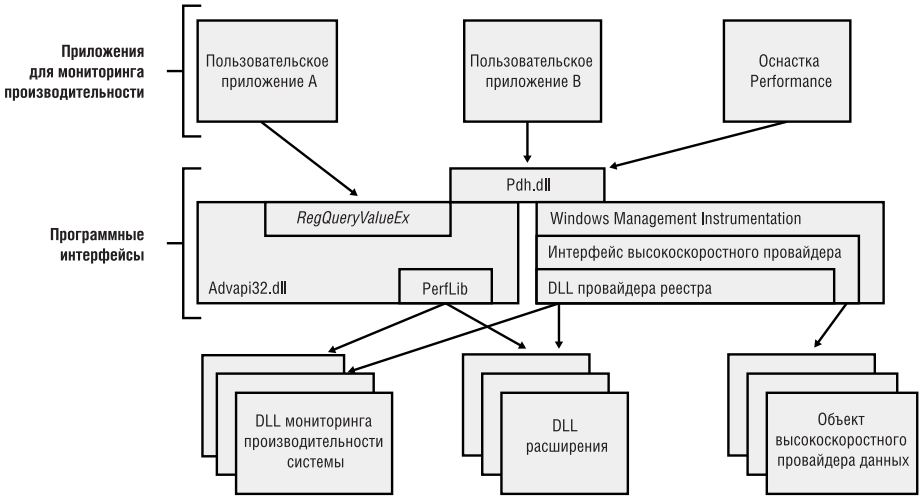


Рис. 4-2. *Архитектура доступа к информации счетчиков производительности*

Анализ и устранение проблем с реестром

Поскольку система и приложения сильно зависят от конфигурационных параметров, изменение данных в реестре может вызвать их сбои. Когда системе или приложению не удается считать параметры, которые, как предполагается, всегда доступны, это программное обеспечение может рухнуть и при этом выводить сообщения об ошибках, скрывающие корень проблемы. Не понимая, как сбоящая система или программа обращается к реестру, практически невозможно выяснить, какие разделы или параметры реестра

сконфигурированы неправильно. В такой ситуации ответ может дать утилита Regmon с сайта *www.sysinternals.com*.

Regmon позволяет наблюдать за обращениями к реестру. При этом Regmon выводит информацию о процессе, обращающемся к реестру, а также сообщает время, тип и результат доступа. Эти сведения полезны для того, чтобы увидеть, как приложения и система взаимодействуют с реестром, найти места хранения конфигурационных параметров, записываемых приложениями и системой, и устранить неполадки, связанные с отсутствием каких-либо разделов или параметров реестра. Regmon включает расширенные средства фильтрации и выделения информации, чтобы вы могли сосредоточиться на операциях над выбранными разделами или параметрами, либо операциях, выполняемых конкретными процессами.

Как работает Regmon

Утилита Regmon полагается на драйвер устройства, который она извлекает из своего исполняемого образа и запускает в период своего выполнения. При первом запуске она требует, чтобы в учетной записи, под которой она выполняется, были привилегии Load Driver и Debug; при последующих запусках в том же сеансе загрузки системы достаточно одной привилегии Debug, так как драйвер является резидентным.

На самом деле внутри исполняемого файла Regmon хранится три драйвера: один — для Windows 95, Windows 98 и Windows Millennium, другой — для Windows NT, Windows 2000 и Windows XP, а третий — для Windows Server 2003. Почему драйвер для Windows Server 2003 отделен от драйвера для аналогичных систем? А потому, что в Windows NT, Windows 2000 и Windows XP единственный способ, которым драйвер может вести мониторинг всех операций с реестром — *перехват системных вызовов* (system-call hooking), и потому, что в Windows Server 2003 драйвер может использовать с той же целью механизм обратного вызова реестра (registry callback mechanism). (Windows 95, Windows 98 и Windows Millennium поддерживают другой механизм мониторинга реестра.)

Вспомните раздел «Диспетчеризация системных сервисов» главы 3 — там говорилось, что адреса функций системных сервисов хранятся в диспетчерской таблице системных сервисов в ядре. Драйвер может обнаруживать вызов системного сервиса, сохранив адрес соответствующей функции из массива и заменив этот элемент массива адресом своей функции-ловушки (hook function). После этого любые вызовы данного сервиса поступают в функцию-ловушку, установленную драйвером, и эта функция может проверять или модифицировать параметры вызова, а при необходимости и выполнять исходную функцию системного сервиса. Если функция-ловушка вызывает исходную функцию, драйвер также получает возможность изучить результат операции и возвращаемые ею данные, например значения параметров реестра. На рис. 4-3 показано, как Regmon перехватывает вызовы функций реестра в режиме ядра.

Рис. 4-3. *Так Regmon использует перехват системных сервисов*

Механизм обратного вызова реестра впервые появился в Windows XP; однако Regmon по-прежнему использует перехват системных вызовов (system-call hooking), работая в Windows XP, потому что в ней этот механизм сообщает не обо всех операциях с реестром. Используя механизм обратного вызова, драйвер регистрирует в диспетчере конфигурации функцию обратного вызова. Диспетчер конфигурации запускает функции обратного вызова, установленные драйвером, в определенные моменты выполнения системных сервисов реестра, чтобы драйвер видел все обращения к реестру и мог их контролировать. Этот механизм используют антивирусные программы, которые сканируют данные реестра или блокируют неавторизованным процессам доступ к реестру для записи.

ЭКСПЕРИМЕНТ: анализ операций с реестром в простаивающей системе

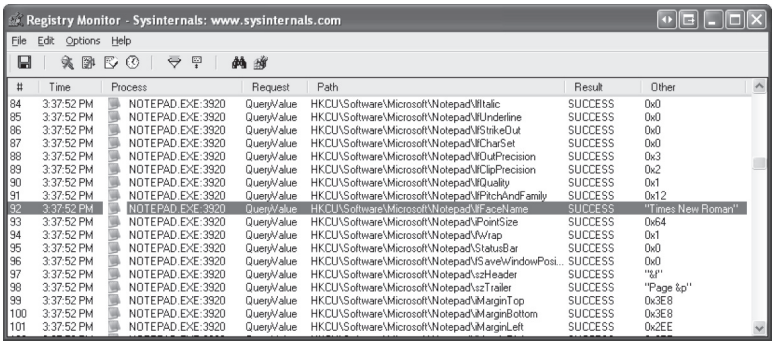
Поскольку реестр реализует функцию *RegNotifyChangeKey*, с помощью которой приложения могут запрашивать уведомление об изменениях в реестре, не опрашивая его постоянно, в простаивающей системе Regmon не должен обнаруживать повторяющиеся обращения к одним и тем же разделам или параметрам реестра. Любая такая активность указывает на плохо написанное приложение, которое отрицательно влияет на общую производительность системы.

Запустите Regmon и через несколько секунд изучите журнал вывода, чтобы выяснить, не пытается ли какая-то программа постоянно опрашивать реестр. Найдя в выводе строку, связанную с опросом, щелкните ее правой кнопкой мыши и выберите из контекстного меню команду Process Properties, чтобы узнать, какой процесс занимается такой деятельностью.

ЭКСПЕРИМЕНТ: поиск параметров приложения в реестре с помощью Regmon

Иногда при анализе проблем нужно определить, где в реестре хранятся те или иные параметры системы или приложения. В этом эксперименте вы используете Regmon для поиска параметров Notepad (Блокнот). Notepad, как и большинство Windows-приложений, сохраняет пользовательские предпочтения (например, включение режима переноса строк, выбранный шрифт и его размер, позиция окна) между запусками. Наблюдая с помощью Regmon, когда Notepad считывает или записывает свои параметры, вы сможете выявить раздел реестра, в котором хранятся эти параметры. Вот как это делается.

1. Пусть Notepad сохранит какой-нибудь параметр, который вы легко найдете в трассировочном выводе Regmon. Для этого запустите Notepad, выберите шрифт Times New Roman и закройте Notepad.
2. Запустите Regmon. Откройте диалоговое окно фильтра выделения информации и введите **notepad.exe** в фильтре Include. Тогда Regmon будет протоколировать только активность notepad.exe в столбце Process или Path.
3. Снова запустите Notepad и остановите в Regmon перехват событий, просто выбрав команду-переключатель Capture Events в меню File утилиты Regmon.
4. Прокрутите полученный журнал к верхней строке и выберите ее.
5. Нажмите Ctrl+F, чтобы открыть диалоговое окно Find, и введите строку поиска **times new**. Regmon должен выделить строку вроде показанной на следующей иллюстрации. Остальные операции в непосредственной близости должны относиться к другим параметрам Notepad.



#	Time	Process	Request	Path	Result	Other
84	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\lfItalic	SUCCESS	0x0
85	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\lfUnderline	SUCCESS	0x0
86	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\lfStrikeOut	SUCCESS	0x0
87	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\lfCharSet	SUCCESS	0x0
88	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\lfOutPrecision	SUCCESS	0x3
89	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\lfClipPrecision	SUCCESS	0x2
90	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\lfQuality	SUCCESS	0x1
91	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\lfPitchAndFamily	SUCCESS	0x12
92	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\lfFaceName	SUCCESS	"Times New Roman"
93	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\lfPointSize	SUCCESS	0x64
94	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\fWrap	SUCCESS	0x1
95	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\StatusBar	SUCCESS	0x0
96	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\fSaveWindowPosi...	SUCCESS	0x0
97	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\szHeader	SUCCESS	"&f"
98	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\szTrailer	SUCCESS	"Page &p"
99	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\iMarginTop	SUCCESS	0x3E8
100	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\iMarginBottom	SUCCESS	0x3E8
101	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\iMarginLeft	SUCCESS	0x2EE

6. Наконец, дважды щелкните выделенную строку. Regmon запустит Regedit (если он еще не выполняется) и заставит его перейти к соответствующему параметру реестра.

Методики анализа проблем с применением Regmon

Выявить причины сбоев приложения или системы, связанные с реестром, позволяют две базовые методики анализа с использованием Regmon.

- Найдите в трассировке Regmon последнее, что делало приложение перед сбоем. Это может указать на источник проблемы.
- Сравните трассировку Regmon для сбойного приложения с аналогичной трассировкой в работающей системе.

При первом подходе запустите сначала Regmon, затем приложение. В момент сбоя вернитесь в Regmon и остановите протоколирование (нажав Ctrl+E). Прокрутите журнал до конца и найдите последние операции, выполнявшиеся приложением перед сбоем (крахом, зависанием или чем-то еще). Начните с последней строки и изучайте, на какие файлы и/или разделы реестра были ссылки, — это часто помогает локализовать источник проблемы.

Второй подход полезен, когда приложение сбоит в одной системе, но работает в другой. Создайте в Regmon журналы трассировки приложения в сбойной и работающей системе, потом откройте их в Microsoft Excel (согласитесь с параметрами по умолчанию, предлагаемыми мастером импорта) и удалите первые три столбца. (Если вы их не удалите, сравнение покажет, что все строки различаются, так как в первых трех столбцах содержится информация, которая меняется между запусками.) Наконец, сравните полученные файлы журналов. (Для этого можно использовать и утилиту WinDiff, которая в Windows XP включена в дистрибутив как один из бесплатных инструментов, а для Windows 2000 предлагается в составе ресурсов.)

Вы должны обратить особое внимание на записи в трассировке Regmon со значениями «NOTFOUND» или «ACCESS DENIED» в столбце Result. NOTFOUND сообщается, когда приложение пыталось обратиться к несуществующему разделу или параметру реестра. Во многих случаях отсутствующий раздел или параметр — вещь безобидная, так как процесс, не сумевший обнаружить искомое в реестре, просто использует значения по умолчанию. Но для некоторых параметров нет значений по умолчанию, и поэтому приложения сбоят, не найдя их в реестре.

Ошибки, связанные с отказом в доступе, — частая причина сбоев приложений; такие ошибки возникают, когда у приложения нет разрешения на доступ к нужному разделу реестра. Это касается приложений, в которых не проверяются результаты операций с реестром или не предусматривается восстановление после соответствующих ошибок.

Также подозрительна строка со значением «BUFROVERFLOW». Она не указывает на наличие в приложении эксплойта (exploit), использующего переполнение буфера. Такое значение посылается диспетчером конфигурации программе, которая выделила под буфер для хранения параметра реестра слишком мало места. Разработчики приложений часто пользуются этим, чтобы определить, какой буфер надо выделить для хранения того или иного значения. Сначала выполняется запрос к реестру с буфером нулевой длины и в ответ поступает сообщение с ошибкой переполнения буфера и ис-

тинным размером данных. Тогда программа создает буфер указанного размера и повторно считывает данные. Поэтому вы должны обнаружить операции, которые возвращают BUFROVERFLOW и при повторной попытке дают успешный результат.

Вот один из примеров использования Regmon для анализа реальной проблемы. Эта утилита избавила пользователя от полной переустановки Windows XP. Симптом был таким: Internet Explorer зависал при запуске, если пользователь предварительно не устанавливал вручную соединение с Интернетом. Оно было задано как соединение по умолчанию, поэтому запуск Internet Explorer должен бы вызывать автоматическое подключение к Интернету (Internet Explorer был настроен на отображение начальной страницы по умолчанию при запуске).

Изучение журнала Regmon для операций Internet Explorer при запуске, начиная с того места, где Internet Explorer зависал, позволило обнаружить запрос, адресованный разделу в HKCU\Software\Microsoft\RAS Phonebook. Пользователь сообщил, что ранее он удалил средство набора телефонных номеров, сопоставленное с этим разделом, и вручную создал соединение по коммутируемой линии. Поскольку имя такого соединения не совпадало с именем удаленной программы, получалось, что соответствующий раздел не был удален программой удаления средства набора телефонных номеров и что именно это было причиной зависания Internet Explorer. После удаления этого раздела Internet Explorer стал работать нормально.

Протоколирование операций под непривилегированными учетными записями или во время входа/выхода

Нередко наблюдается следующая ситуация. Приложение работает при выполнении под учетной записью, входящей в группу Administrators (Администраторы), и сбоит при запуске под учетной записью непривилегированного пользователя. Как уже говорилось, Regmon требует привилегий, которые обычно не выдаются стандартным учетным записям пользователей, но вести трассировку приложений, выполняемых в сеансе входа непривилегированного пользователя, все же можно. Для этого запустите Regmon под административной учетной записью командой *runas*.

Если проблема с реестром относится ко входу или выходу по учетной записи, вы также должны предпринять особые меры, чтобы использовать Regmon для трассировки этих этапов сеанса входа. Приложения, выполняемые под системной учетной записью, не завершаются при выходе пользователя, и благодаря этому вы можете работать с Regmon, несмотря на выход текущего пользователя и последующий вход того же или другого пользователя. Чтобы запустить Regmon под системной учетной записью, введите команду *at*, встроенную в Windows, и укажите флаг */interactive* или запустите утилиту PsExec с сайта *www.sysinternals.com*, например так:

```
psexec -i -s -d c:\regmon.exe
```

Ключ *–i* сообщает PsExec, что окно Regmon должно появиться в интерактивной консоли, ключ *–s* заставляет PsExec запустить Regmon под системной учетной записью, а ключ *–d* указывает PsExec запустить Regmon и завершиться, не дожидаясь закрытия Regmon. После этой команды данный экземпляр Regmon переживет выход пользователя, и его окно вновь появится на рабочем столе, когда кто-то войдет в систему; при этом он будет протоколировать активность в реестре в обоих случаях.

Еще один способ мониторинга активности в реестре во время входа, выхода, загрузки системы или ее выключения — использовать функцию Regmon для протоколирования с момента загрузки системы. Для этого вы должны выбрать Log Boot в меню Options. При следующем запуске системы драйвер устройства Regmon будет протоколировать активность в реестре с самых ранних этапов загрузки, записывая информацию в журнал \Windows\Regmon.log. Протоколирование будет продолжаться до тех пор, пока не закончится свободное место на диске, пока система не будет выключена или пока вы не запустите Regmon. Файл журнала, хранящий трассировку операций над реестром при загрузке, входе, выходе и выключении системы Windows XP, обычно занимает 50–150 Мб.

Внутренние механизмы реестра

Из этого раздела вы узнаете, как диспетчер конфигурации (компонент исполнительной системы, реализующий реестр) организует файлы реестра на диске. Мы исследуем, как диспетчер конфигурации управляет реестром по мере его чтения и изменения приложениями и другими компонентами системы. Мы также обсудим механизмы, с помощью которых диспетчер конфигурации позволяет восстанавливать реестр, даже если система рухнула непосредственно в ходе внесения в него изменений.

Кусты

Реестр представлен на диске не просто одним большим файлом, а набором отдельных файлов, называемых *кустами* (hives). В каждом кусте содержится дерево реестра, у которого есть раздел, служащий корнем, или начальной точкой, дерева. Подразделы с их параметрами находятся под корнем. Возможно, вы подумали, что корневые разделы, показываемые редактором реестра, соответствуют корневым разделам кустов, но это не так. В таблице 4-5 перечислены кусты реестра и имена их файлов на диске. Полные имена всех файлов кустов (вместе с путями), кроме относящихся к профилям пользователей, жестко определяются самим диспетчером конфигурации. При загрузке кустов диспетчер конфигурации отмечает путь к каждому кусту в подразделе HKLM\SYSTEM\CurrentControlSet\Control\Hivelist и удаляет пути к выгруженным из памяти кустам. (Профили пользователей выгружаются в отсутствие ссылок на них.) Для формирования привычной структуры реестра, отображаемой редактором реестра, диспетчер конфигурации создает корневые разделы и связывает кусты друг с другом.

Таблица 4-5. *Дисковые файлы, соответствующие путям в реестре*

Путь к кусту в реестре	Путь к файлу куста
HKEY_LOCAL_MACHINE\SYSTEM	\Windows\System32\Config\System
HKEY_LOCAL_MACHINE\SAM	\Windows\System32\Config\Sam
HKEY_LOCAL_MACHINE\SECURITY	\Windows\System32\Config\Security
HKEY_LOCAL_MACHINE\SOFTWARE	\Windows\System32\Config\Software
HKEY_LOCAL_MACHINE\HARDWARE	Изменяемый (volatile) куст
HKEY_LOCAL_MACHINE\SYSTEM\Clone	Изменяемый куст (только в Windows 2000)
HKEY_USERS\<*SID_для_пользователя*>	\Documents and Settings\ <*имя_ пользователя*>\Ntuser.dat
HKEY_USERS\<*SID_для_пользователя*> _Classes	\Documents and Settings\ <*имя_пользователя*>\Local Settings\ Application Data\Microsoft\Windows\ Usrclass.dat
HKEY_USERS\.DEFAULT	\Windows\System32\Config\Default

Заметьте, что некоторые кусты, перечисленные в таблице 4-5, являются изменяемыми и не имеют сопоставленных файлов. Система создает и манипулирует такими кустами только в памяти, поэтому они существуют лишь временно. Изменяемые кусты создаются при каждой загрузке системы. Пример подобного куста — HKLM\HARDWARE, в котором хранятся сведения о физических устройствах и назначенных им ресурсах. Распознавание оборудования и распределение ресурсов происходят при каждой загрузке системы, поэтому было бы нелогично хранить данные этого куста на диске.

ЭКСПЕРИМЕНТ: загрузка и выгрузка кустов вручную

Regedt32 в Windows 2000 и Regedit в Windows XP или Windows Server 2003 позволяют загружать кусты, к которым можно обращаться через меню File этих редакторов реестра. Такая возможность полезна при анализе проблем, когда нужно просмотреть или отредактировать куст, полученный с незагружаемой системы или из резервной копии. В этом эксперименте вы используете Regedt32 (при наличии Windows 2000) или Regedit (при наличии Windows XP или Windows Server 2003) для загрузки версии куста HKLM\SYSTEM, создаваемой программой Windows Setup и сохраняемый в каталоге \Windows\Repair в ходе установки.

1. Кусты можно загружать только в HKLM или HKU, поэтому откройте Regedit или Regedt32, укажите HKLM, а затем выберите Load Hive (Загрузить куст) из меню File (Файл) в Regedit или из меню Registry (Реестр) в Regedt32.
2. Перейдите в каталог \Windows\Repair в диалоговом окне Load Hive (Загрузить куст), выберите System.bak и откройте его. При запросе введите **Test** в качестве имени раздела, в который будет загружаться этот куст.

3. Откройте только что созданный раздел HKLM\Test и изучите содержимое куста.
4. Откройте HKLM\System\CurrentControlSet\Control\Hivelist и найдите элемент \Registry\Machine\Test, который продемонстрирует, как диспетчер конфигурации перечисляет загруженные кусты в разделе HiveList.
5. Укажите HKLM\Test и выберите Unload Hive (Выгрузить куст) из меню File в Regedit или из меню Registry в Regedt32 для выгрузки этого куста.

Лимиты на размеры кустов

В некоторых случаях размеры кустов ограничиваются. Например, Windows ограничивает размер куста HKLM\SYSTEM. Это делается из-за того, что Ntldr считывает весь куст HKLM\SYSTEM в физическую память почти в самом начале процесса загрузки, когда поддержки виртуальной памяти еще нет. Кроме того, Ntldr загружает в физическую память Ntoskrnl и драйверы устройств периода загрузки. (Подробнее о роли Ntldr в процессе загрузки см. главу 6.) В Windows 2000 Ntldr устанавливает фиксированный верхний предел на размер этого куста в 12 Мб, но в Windows XP и Windows Server 2003 тот же куст может быть размером до 200 Мб или до четверти объема физической памяти, установленной в системе (в зависимости от того, какой предел будет достигнут раньше).

В Windows 2000 также существует лимит на общий размер всех загруженных кустов. Она использует для хранения кустов реестра пул подкачиваемой памяти, и поэтому общий объем загруженных данных реестра ограничен доступным размером этого пула. При инициализации диспетчер памяти определяет его размер на основе целого ряда факторов, в том числе объема физической памяти в системе. В системе, где диспетчер памяти создает самый большой из возможных пул подкачиваемой памяти, размер реестра ограничен 376 Мб. Поскольку система не сможет эффективно работать, если пула подкачиваемой памяти будет недостаточно для других целей, Windows 2000 не позволит данным реестра занять более 80% этого пула. Для просмотра или модификации ограничения на размер реестра, как показано на рис. 4-4, щелкните кнопку Change (Изменить) в разделе Virtual Memory (Виртуальная память) диалогового окна Performance Options (Параметры быстродействия), доступного с вкладки Advanced (Дополнительно) окна свойств системы.

Лимит на общий размер загруженных кустов реестра может привести к ограничению числа пользователей, одновременно входящих в систему Windows 2000 с Terminal Services, поскольку каждый профиль пользователя увеличивает размер загруженных кустов. В Windows XP и Windows Server 2003 диспетчер конфигурации использует не пул подкачиваемой памяти, а функции проецирования в системную память, предоставляемые диспетчером памяти. При этом проецируются лишь те части кустов реестра, к которым происходят обращения в данный момент времени. Ограничений на размер

реестра в Windows XP или Windows Server 2003 нет, и общий размер загруженных кустов не сказывается на масштабируемости Terminal Services.

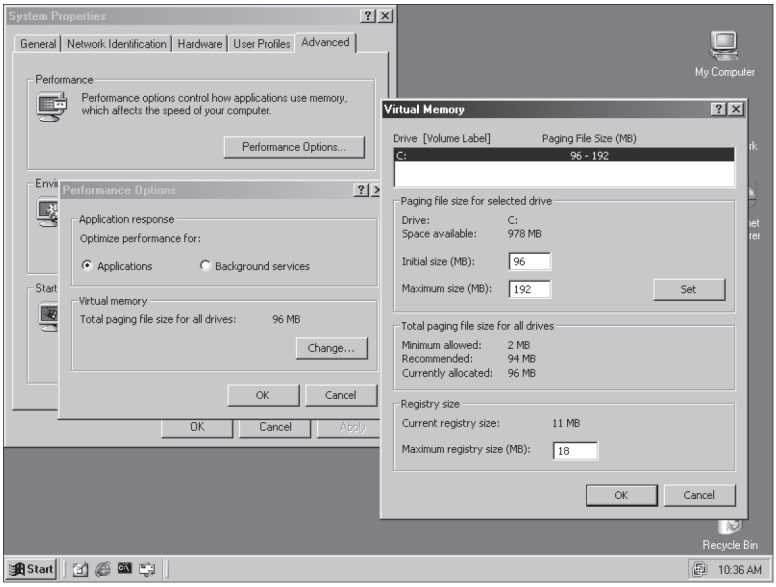
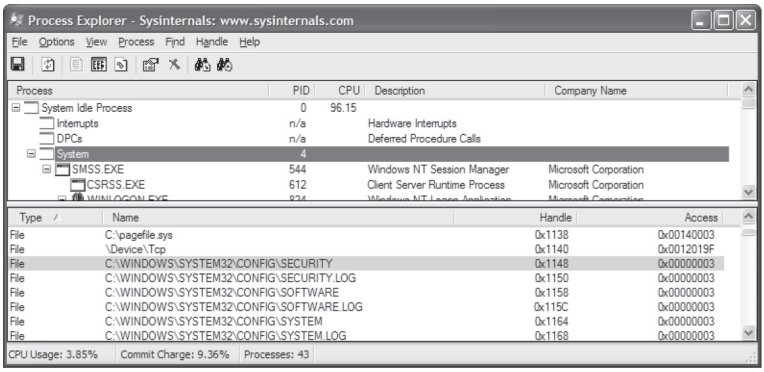


Рис. 4-4. *Задание лимита на размер реестра в Windows 2000*

ЭКСПЕРИМЕНТ: просмотр описателей кустов

Диспетчер конфигурации открывает кусты, используя таблицу описателей режима ядра (см. главу 3), поэтому он может обращаться к ним из контекста любого процесса. Применение такой таблицы — эффективная альтернатива подходу, основанному на использовании драйверов или компонентов исполнительной системы для простого обращения из системных процессов к одним лишь описателям (которые должны быть защищены от пользовательских процессов). Просмотреть описатели кустов можно с помощью утилиты Process Explorer (*wwwsysinternals.com*). В Windows 2000 диспетчер объектов сообщает об описателях из таблицы как об открытых в системном процессе System Idle, а в Windows XP и Windows Server 2003 он показывает описатели как открытые в процессе System. Укажите нужный процесс и выберите Handles из подменю Lower Pane View в меню View. Задайте сортировку по типу описателя и прокручивайте список, пока не увидите файлы кустов, как на следующей иллюстрации.



Особый тип разделов, *символьная ссылка* (symbolic link), позволяет диспетчеру конфигурации связывать кусты для организации реестра. Символьная ссылка — это раздел, который переадресует диспетчер конфигурации к другому разделу. Так, раздел HKLM\SAM представляет собой символьную ссылку на раздел в корне куста SAM.

Структура куста

Диспетчер конфигурации делит куст на логические единицы, называемые *блоками* (blocks), по аналогии с тем, как файловая система делит диск на кластеры. По определению размер блока реестра составляет 4096 байтов (4 Кб). Размер куста увеличивается кратно размеру блоков. Первый блок куста называется *базовым* (base block); он включает глобальную информацию о кусте, в том числе сигнатуру *regf*, идентифицирующую файл как куст, порядковые номера, метку времени последней записи в куст, номер версии формата, контрольную сумму и внутреннее имя файла куста (например, *\Устройство\Раздел_жесткого_диска1*\WINDOWS\SYSTEM32\Config\SAM). Мы поясним смысл порядковых номеров и метки времени, когда будем рассматривать механизм записи данных в файл куста. Номер версии формата указывает формат данных куста. В Windows 2000 диспетчер конфигурации использует формат данных куста версии 1.3. В Windows XP и Windows Server 2003 применяется тот же формат данных для совместимости с профилями роуминга Windows 2000, но для кустов System и Software используется формат версии 1.5, обеспечивающий более эффективный поиск, а также хранение больших значений.

Windows упорядочивает хранимые в кусте данные с помощью контейнеров, которые называются *ячейками* (cells). Ячейка может содержать раздел, параметр, дескриптор защиты, список подразделов или параметров раздела. Поле в начале ячейки описывает тип ее данных. Все типы данных поясняются в таблице 4-6. Размер ячейки указывается в ее заголовке. Когда ячейка присоединяется к кусту, последний должен быть соответственно увеличен. Для этого система создает блок, называемый *приемником* (bin). Размер приемника равен размеру ячейки, округленному до ближайшего большего значения, кратного размеру блока. Пространство между концом ячейки и концом при-

емника считается свободным, и система может помещать в него другие ячейки. Приемники тоже имеют заголовки, но с сигнатурой *hbin*, и поле, в которое записывается размер приемника и его смещение в файле куста.

Таблица 4-6. *Типы данных в ячейках*

Тип данных	Описание
Ячейка раздела (key cell)	Ячейка, содержащая раздел реестра; также называется *узлом раздела* (key node). Включает сигнатуру (*kn* — раздел, *kl* — символьная ссылка), метку времени последнего обновления, имя раздела (например, CurrentControlSet) и следующие индексы: родительской ячейки того же типа, ячейки списка подразделов данного раздела, ячейки дескриптора защиты раздела и строки с именем класса раздела
Ячейка параметра (value cell)	Ячейка, содержащая информацию о параметре раздела. Включает сигнатуру *kv*, тип параметра (например, REG_DWORD или REG_BINARY) и его имя (скажем, Boot-Execute). Ячейка этого типа также содержит индекс ячейки с данными параметра
Ячейка списка подразделов (subkey-list cell)	Ячейка, состоящая из списка индексов для ячеек разделов, которые являются подразделами общего родительского раздела
Ячейка списка параметров (value-list cell)	Ячейка, состоящая из списка индексов для ячеек параметров, которые являются параметрами общего родительского раздела
Ячейка дескриптора защиты (security-descriptor cell)	Ячейка с дескриптором защиты. Включает сигнатуру (*ks*) и счетчик ссылок, регистрирующий число узлов разделов, использующих этот дескриптор. Ячейки разделов могут совместно использовать ячейки дескрипторов защиты

Используя для отслеживания активных частей реестра приемники вместо ячеек, Windows упрощает себе управление реестром. Так, система обычно создает и удаляет приемники реже, чем ячейки, а это позволяет диспетчеру конфигурации эффективнее управлять памятью. Считывая куст в память, диспетчер конфигурации может выбирать только приемники, содержащие ячейки (т. е. активные приемники), и игнорировать пустые (удаленные). В результате добавления или удаления ячеек куст может содержать пустые приемники вперемешку с активными. Такая ситуация напоминает фрагментацию диска, возникающую при создании и удалении файлов. Когда приемник становится пустым, диспетчер конфигурации объединяет его со смежными пустыми приемниками, формируя непрерывный пустой приемник как можно большего размера. Диспетчер конфигурации также объединяет смежные пустые ячейки для формирования свободных ячеек большего размера. (Диспетчер конфигурации уплотняет куст, только когда приемники в конце куста освобождаются. Вы можете уплотнить реестр за счет его резервного копирования и последующего восстановления с помощью Windows-функций *RegSaveKey* и *RegReplaceKey*, используемых утилитой Windows Backup.)

Ссылки, образующие структуру куста, называются *индексами ячеек* (cell indexes). Индекс ячейки представляет собой ее смещение в файле куста. Таким образом, он похож на указатель из одной ячейки на другую и интерпретируется диспетчером конфигурации относительно начала куста. Например, как видно из таблицы 4-6, ячейка раздела содержит поле с индексом ячейки родительского раздела; индекс ячейки подраздела указывает на ячейку со списком подчиненных ему подразделов. Ячейка списка подразделов содержит список индексов, ссылающихся на ячейки подчиненных подразделов. Поэтому если вам нужно найти, скажем, ячейку раздела для подраздела A, родительским разделом которого является раздел B, вы должны сначала найти ячейку со списком подразделов раздела B по ее индексу в ячейке раздела B. После этого с помощью списка индексов из ячейки списка подразделов раздела B можно отыскать ячейки любых подразделов раздела B. При этом для каждой ячейки подраздела вы проверяете, не совпадает ли хранящееся там имя раздела с именем искомого (в данном случае — A).

Ячейки, приемники и блоки можно легко перепутать, поэтому для прояснения различий между ними обратимся к структуре простого куста реестра. Образец файла куста реестра, схема которого показана на рис. 4-5, включает в себя базовый блок и два приемника. Первый приемник пуст, а во втором есть несколько ячеек. Логично, что в таком кусте может быть всего два раздела: корневой Root и его подраздел, Sub Key. В Root находятся два параметра: Val 1 и Val 2. Ячейка списка подразделов определяет местонахождение подразделов корневого раздела, а ячейка списка параметров — адрес параметров корневого раздела. Свободные промежутки во втором приемнике являются пустыми ячейками. Учтите, что на схеме не показаны ячейки дескрипторов защиты для двух разделов, которые должны присутствовать в составе куста.

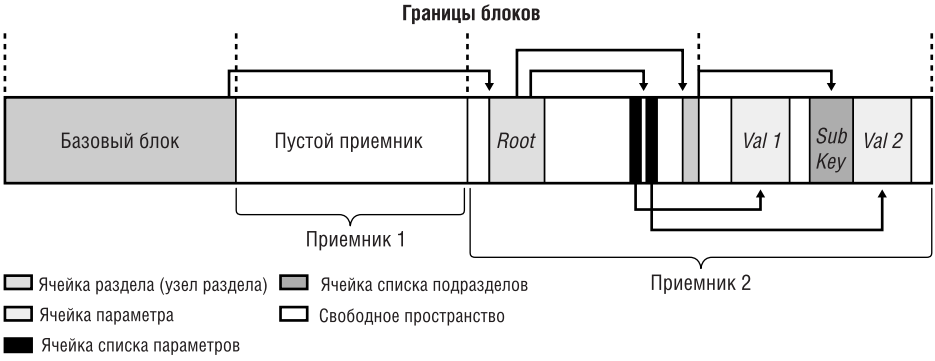


Рис. 4-5. *Внутренняя структура куста реестра*

На рис. 4-6 показано окно утилиты Disk Probe (Dskprobe.exe) с образцом содержимого первого приемника куста SYSTEM. Обратите внимание на сигнатуру приемника, *hbin*. Под ней можно увидеть сигнатуру *nk*. Это сигнатура ячейки раздела (*kn*). Обратный порядок отображения сигнатуры определяется

порядком хранения данных в системах типа x86. Ячейка, которой диспетчер конфигурации присвоил внутреннее имя $$$PROTO.HIV, является корневой ячейкой куста SYSTEM, как указывает следующее за сигнатурой *nk* имя.

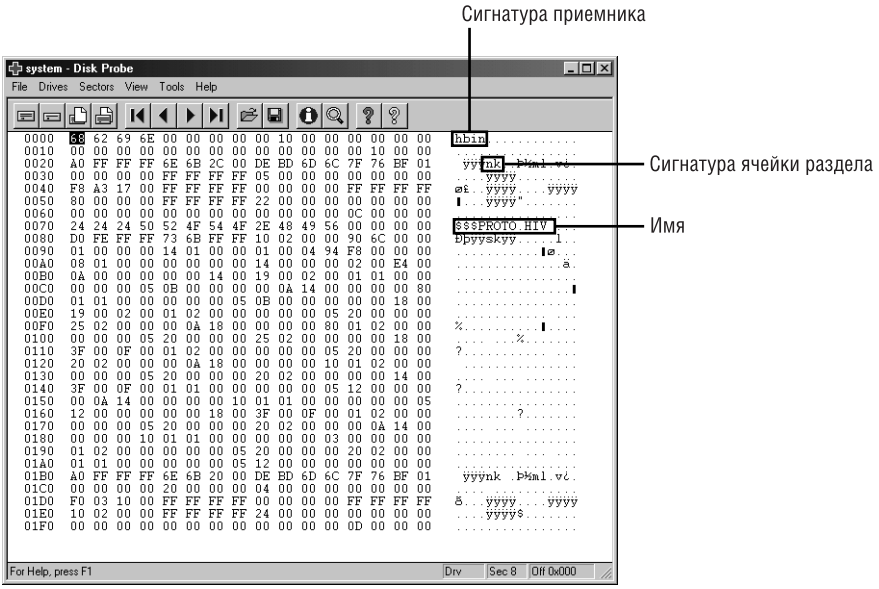


Рис. 4-6. *Двоичное содержимое первого приемника в кусте SYSTEM*

Для оптимизации поиска подразделов и параметров диспетчер конфигурации сортирует ячейки списков подразделов в алфавитном порядке. Если нужно найти подраздел в списке, диспетчер использует двоичный поиск. При этом он сразу обращается в середину списка. Если искомое имя в соответствии с алфавитным порядком находится перед разделами из середины списка, диспетчер узнает, что оно хранится в первой половине списка. В ином случае оно должно быть во второй половине списка подразделов. И так до тех пор, пока диспетчер не найдет искомый подраздел или не обнаружит его отсутствие. Ячейки списков параметров не сортируются, так что новые параметры всегда добавляются в конец списка.

Карты ячеек

Диспетчер конфигурации не обращается к файлам кустов на диске при каждом обращении к реестру. Windows хранит в адресном пространстве ядра версию каждого куста. При инициализации куста диспетчер конфигурации определяет размер его файла, выделяет из подкачиваемого пула нужный объем памяти и считывает файл куста в память (о пуле подкачиваемой памяти см. главу 7). Поскольку все загруженные кусты реестра хранятся в подкачиваемом пуле, в Windows 2000 они, как правило, занимают его наибольшую часть. (Для исследования этого пула используйте утилиту Poolmon, описываемую в одном из экспериментов главы 7.)

В Windows XP и Windows Server 2003 диспетчер конфигурации проецирует части куста в память по мере того, как возникает необходимость в доступе к ним. При этом он обращается к функциям проецирования файлов в диспетчере кэша для отображения 16-килобайтных представлений на файлы кустов (о диспетчере кэша см. главу 10). Чтобы проекция куста не заняла весь адресный диапазон диспетчера кэша, диспетчер конфигурации пытается хранить не более 256 представлений куста в любой момент времени, отменяя проецирование реже всего используемых представлений по достижении этого предела. Диспетчер конфигурации по-прежнему использует подкачиваемый пул для хранения различных структур данных, но занимает в нем лишь малую часть по сравнению с Windows 2000.

ПРИМЕЧАНИЕ В Windows XP и Windows Server 2003 диспетчер конфигурации сохраняет блок в подкачиваемом пуле вместо его проецирования, если размер этого блока превышает 256 Кб.

Если бы размер кустов никогда не увеличивался, диспетчер конфигурации мог бы выполнять все операции с копией реестра в памяти, как с обыкновенным файлом. Зная индекс ячейки, диспетчер конфигурации мог бы вычислить ее адрес в памяти, просто сложив индекс ячейки, представляющий смещение в файле куста, с базовым адресом копии куста в памяти. Именно так и поступает Ntldr с кустом SYSTEM на ранних этапах загрузки: он полностью считывает его в память как неизменяемый и для поиска нужных ячеек суммирует их индексы с базовым адресом копии куста в памяти. К сожалению, по мере появления новых разделов и параметров кусты разрастаются, а это означает, что система должна выделять дополнительную память из подкачиваемого пула для хранения новых приемников с добавляемыми разделами и параметрами. Так что данные реестра не обязательно хранятся в непрерывной области подкачиваемой памяти.

ЭКСПЕРИМЕНТ: наблюдение за использованием пула подкачиваемой памяти для кустов реестра

Административных утилит, которые показывали бы объем памяти из подкачиваемого пула, используемой кустами реестра вместе с профилями пользователей, в Windows 2000 нет. Однако команда *!reg dumppool* отладчика ядра сообщает не только число страниц, задействованных каждым загруженным кустом, но и количество страниц, занятых постоянными и переменными данными. В конце отчета команда выводит суммарный объем памяти, занятой кустами реестра. (Учтите, что эта команда показывает лишь последние 32 символа в имени куста.)

```
kd> !reg dumppool

dumping hive at e20d66a8 (a\Microsoft\Windows\UsrClass.dat)
    Stable Length = 1000
    1/1 pages present
    Volatile Length = 0
```

см. след. стр.

```
dumping hive at e215ee88 (ettings\Administrator\ntuser.dat)
    Stable Length = f2000
    242/242 pages present
    Volatile Length = 2000
    2/2 pages present

dumping hive at e13fa188 (\SystemRoot\System32\Config\SAM)
    Stable Length = 5000
    5/5 pages present
    Volatile Length = 0
...
```

ЭКСПЕРИМЕНТ: наблюдение за использованием памяти кустами

В Windows XP и Windows Server 2003 статистику по использованию памяти кустами реестра можно просмотреть командой *!reg hivelist*. Вывод команды включает размеры постоянных (сохраняемых на диске) и переменных данных, число активных представлений и количество представлений, заблокированных в памяти:

```
------------------------------------------------------------------------------
------------------------
| HiveAddr |Stable Length|Stable Map|Volatile Length|Volatile
Map|MappedViews|PinnedViews|U(Cnt)| BaseBlock | FileName
------------------------------------------------------------------------------
------------------------
| e22f8b68 |       5000  | e22f8bc4 |       1000    |  e22f8ca0  |        2
|        0  |     0| e2353000  | \Microsoft
\Windows\UsrClass.dat
| e28c3008 |     3fe000  | e1e84000 |       c000    |  e28c3140  |      116
|        0  |     0| e1e48000  | ttings\Adm
inistrator\ntuser.dat
| e23ec008 |       1000  | e23ec064 |          0    |  00000000  |        1
|        0  |     0| e23ee000  | \Microsoft
\Windows\UsrClass.dat
| e23ed760 |      37000  | e23ed7bc |       1000    |  e23ed898  |       14
|        0  |     0| e23ef000  | ettings\Lo
calService\ntuser.dat
...
```

Здесь куст профиля для учетной записи Administrator (полный путь к которому, \Documents and Settings\Administrator\ntuser.dat, в выводе обрезан) имеет 116 спроецированных представлений и размер около 4 Мб (0x3f000 в шестнадцатеричной форме). Команда *!reg viewlist* показывает спроецированные представления заданного куста. Вот как выглядит вывод этой команды при ее выполнении применительно к кусту UsrClass.dat, который был показан как первый куст в выводе команды *!reg hivelist*:

```
kd> !reg viewlist e22f8b68

   0  Pinned Views ; PinViewListHead = e22f8da0 e22f8da0

   2  Mapped Views ; LRUViewListHead = e1cf4448 e1c5d440
-------------------------------------------------------------------------------
-------------------------
| ViewAddr |FileOffset|   Size   |ViewAddress|   Bcb    |    LRUViewList
|    PinViewList     | UseCount |
-------------------------------
-------------------------
| e1cf4448 |        0 |     4000 | c9a40000  | 8a4bb0e9 | e1c5d440
e22f8d98 | e1cf4450  e1cf4450 |        0 |
| e1c5d440 |     4000 |     2000 | c9a44000  | 8a4bb0e9 | e22f8d98
e1cf4448 | e1c5d448  e1c5d448 |        0 |
-------------------------------------------------------------------------------
-------------------------
```

В этом выводе показаны адреса двух представлений, которые команда *hivelist* сообщила для куста в столбце ViewAddress. Используя команду *db* отладчика ядра для получения содержимого памяти по адресу первого представления, вы обнаружите, что это проекция базового блока куста (она распознается по сигнатуре *regf*):

```
kd> db c9a40000
c9a40000  72 65 67 66 d5 01 00 00-d5 01 00 00 cc 20 43 c7  regf......... C.
c9a40010  3d 40 c4 01 01 00 00 00-03 00 00 00 00 00 00 00  =@..............
c9a40020  01 00 00 00 20 00 00 00-00 50 00 00 01 00 00 00  .... ....P......
c9a40030  5c 00 4d 00 69 00 63 00-72 00 6f 00 73 00 6f 00  \.M.i.c.r.o.s.o.
c9a40040  66 00 74 00 5c 00 57 00-69 00 6e 00 64 00 6f 00  f.t.\.W.i.n.d.o.
c9a40050  77 00 73 00 5c 00 55 00-73 00 72 00 43 00 6c 00  w.s.\.U.s.r.C.l.
c9a40060  61 00 73 00 73 00 2e 00-64 00 61 00 74 00 00 00  a.s.s...d.a.t...
c9a40070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

Для работы с дискретными адресами, ссылающимися на данные кустов в памяти, диспетчер конфигурации использует стратегию, аналогичную применяемой диспетчером памяти Windows для проецирования виртуальных адресов памяти на физические. Двухуровневая схема, принятая для диспетчера конфигурации, показана на рис. 4-7. На входе принимается индекс ячейки (т. е. смещение в файле куста), а на выходе возвращается адрес блока, в который попадает индекс данной ячейки, и адрес блока, в котором находится указанная ячейка. Вспомните, что в приемнике может быть более одного блока, а куст разрастается за счет увеличения размеров приемников. Ввиду этого Windows всегда отводит под приемник непрерывную область памяти, и все его блоки находятся в одном и том же представлении, принадлежащем диспетчеру кэша (в Windows XP и Windows Server 2003), или в одной и той же части пула подкачиваемой памяти.

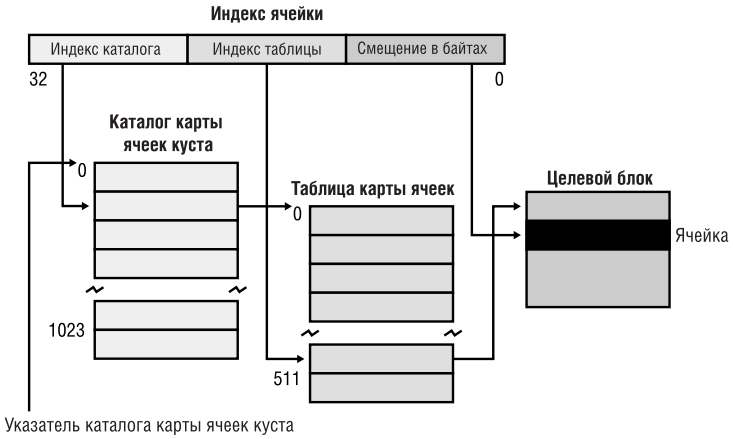


Рис. 4-7. *Структура индекса ячейки*

Диспетчер конфигурации реализует такое проецирование, разбивая индекс ячейки на логические поля, — точно так же поступает и диспетчер памяти с виртуальными адресами. Windows интерпретирует первое поле индекса ячейки как индекс каталога карты ячеек куста. В каталоге карты ячеек имеется 1024 элемента, каждый из которых ссылается на таблицу карты ячеек, а каждая таблица в свою очередь содержит 512 элементов. Элемент в таблице карты ячеек определяется вторым полем индекса ячейки. В этом элементе содержатся адреса приемника и блоков с ячейкой в памяти. В Windows XP и Windows Server 2003 не все приемники обязательно проецируются в память, и, если поиск ячейки дает нулевой адрес, диспетчер конфигурации проецирует приемник в память, при необходимости отменяя проецирование другого приемника из своего списка.

На завершающем этапе процесса проецирования диспетчер конфигурации интерпретирует последнее поле индекса ячейки как смещение в найденном блоке для определения точного местонахождения ячейки в памяти. При инициализации куста диспетчер конфигурации динамически создает таблицы сопоставлений с записями для всех блоков куста, а в дальнейшем добавляет и удаляет таблицы из каталога ячеек по мере изменения размера куста.

Пространство имен и механизмы работы реестра

Для интеграции пространства имен реестра с общим пространством имен ядра диспетчер конфигурации определяет тип объектов «раздел реестра». Он помещает такой объект с именем Registry в корень пространства имен Windows и использует его как точку входа в реестр. Regedit показывает имена разделов в виде HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet, но подсистема Windows транслирует эти имена в соответствии с пространством имен своих объектов (например, \Registry\Machine\System\CurrentControlSet). Диспетчер объектов, анализируя подобное имя, распознает ссылку на объект Registry и тут же передает остальную часть имени диспетчеру конфи-

гурации. Последний берет на себя дальнейший разбор имени, просматривая свое внутреннее дерево куста для поиска нужного раздела или параметра. Прежде чем описывать последовательность действий при типичной операции с реестром, нужно обсудить объекты «раздел реестра» и *блоки управления разделом* (key control blocks). Всякий раз, когда программа создает или открывает раздел реестра, диспетчер объектов передает ей описатель для ссылки на этот раздел. Описатель соответствует объекту «раздел реестра», созданному диспетчером конфигурации с участием диспетчера объектов. Опираясь на диспетчер объектов, диспетчер конфигурации использует все предоставляемые им преимущества в защите объектов и учете ссылок.

Для каждого открытого раздела реестра диспетчер конфигурации создает блок управления разделом. В таком блоке хранится полный путь раздела, индекс ячейки узла раздела, к которому относится данный блок, и флаг, уведомляющий диспетчер конфигурации, надо ли удалять ячейку раздела (на которую ссылается данный блок) после закрытия последнего описателя раздела. Windows помещает все блоки управления разделами в хэш-таблицу, что обеспечивает быстрый поиск нужного блока по имени. Объекты «раздел реестра» указывают на соответствующие блоки управления, и, если два приложения открывают один и тот же раздел реестра, каждое получает свой объект, указывающий на общий блок управления.

Приложение, открывая существующий раздел реестра, начинает с того, что сообщает его имя API-функции реестра, которая вызывает процедуру разбора имени, принадлежащую диспетчеру объектов. Найдя нужный объект «раздел реестра» в пространстве имен диспетчера конфигурации, диспетчер объектов возвращает ему полученный путь. Диспетчер конфигурации, используя структуры данных куста, содержащиеся в памяти, ищет указанный раздел среди всех разделов и подразделов. Если он находит ячейку раздела, поиск продолжается в дереве блоков управления разделами, что позволяет узнать, открыт ли данный раздел (тем же или другим приложением). Процедура поиска оптимизирована так, чтобы поиск всегда начинался с ближайшего предка с уже открытым блоком управления. Например, если приложение открывает \Registry\Machine\Key1\Subkey2 в то время, как \Registry\Machine уже открыт, то процедура разбора в качестве отправной точки использует блок управления разделом \Registry\Machine. Если раздел открыт, диспетчер конфигурации увеличивает счетчик ссылок в блоке управления этим разделом. В ином случае диспетчер конфигурации создает новый блок управления и помещает его в дерево. Далее диспетчер конфигурации создает объект «раздел реестра», передает указатель на него блоку управления разделом и возвращает управление диспетчеру объектов, который передает приложению описатель.

Когда приложение создает новый раздел реестра, диспетчер конфигурации сначала ищет для нового раздела ячейку родительского раздела. Далее он находит список свободных ячеек куста, в котором будет находиться новый раздел, и определяет, есть ли достаточно большие ячейки для размещения ячейки нового раздела. Если таковых нет, диспетчер конфигурации создает новый приемник и помещает в него ячейку, а остальное свободное

пространство приемника регистрирует в списке свободных ячеек. Новая ячейка раздела заполняется соответствующей информацией, включая имя раздела. Диспетчер конфигурации добавляет ячейку раздела в список подразделов ячейки родительского раздела. Наконец, система сохраняет в новой ячейке индекс родительской ячейки.

Диспетчер конфигурации использует счетчик ссылок блока управления разделом для определения момента его удаления. Когда закрываются все описатели, счетчик ссылок обнуляется, и это говорит о том, что данный блок управления разделом больше не нужен. Если приложение, вызывающее API-функцию для удаления раздела, устанавливает флаг удаления, диспетчер конфигурации может удалить соответствующий раздел куста, так как он больше не используется ни одним приложением.

ЭКСПЕРИМЕНТ: просмотр блоков управления разделами

Команда *!reg openkeys* отладчика ядра позволяет перечислить все блоки управления разделами, созданные в системе. В качестве альтернативы, если вы хотите просмотреть блок управления разделом для конкретного открытого раздела, используйте *!reg findkcb*:

```
kd> !reg findkcb \registry\machine\software\microsoft

Found KCB = e1034d40 :: \REGISTRY\MACHINE\SOFTWARE\MICROSOFT
```

Для анализа блока управления разделом, о котором сообщила предыдущая команда, предназначена команда *!reg kcb*:

```
kd> !reg kcb e1034d40

Key              : \REGISTRY\MACHINE\SOFTWARE\MICROSOFT
RefCount         : 1f
Flags            : CompressedName, Stable
ExtFlags         :
Parent           : 0xe1997368
KeyHive          : 0xe1c8a768
KeyCell          : 0x64e598 [cell index]
TotalLevels      : 4
DelayedCloseIndex: 2048
MaxNameLen       : 0x3c
MaxValueNameLen  : 0x0
MaxValueDataLen  : 0x0
LastWriteTime    : 0x 1c42501:0x7eb6d470
KeyBodyListHead  : 0xe1034d70 0xe1034d70
SubKeyCount      : 137
ValueCache.Count : 0
KCBLock          : 0xe1034d40
KeyLock          : 0xe1034d40
```

Поле Flags указывает, что имя хранится в сжатой форме, а поле SubKeyCount — что в разделе имеется 137 подразделов.

Надежность хранения данных реестра

Для обеспечения гарантированной возможности восстановления постоянных кустов реестра (т. е. кустов, которым соответствуют файлы на диске) диспетчер конфигурации использует *регистрационные кусты* (log hives). С каждым постоянным кустом сопоставлен регистрационный, представляющий собой скрытый файл с именем куста и расширением LOG. Так, в каталоге \Windows\System32\Config присутствуют System.log, Sam.log и другие LOG-файлы. При инициализации куста диспетчер конфигурации создает битовый массив, в котором каждый бит представляет часть куста размером 512 байтов — *сектор куста* (hive sector). Поэтому и массив называется *массивом измененных секторов* (dirty sector array). Установленный бит этого массива указывает на то, что соответствующий сектор куста в памяти изменен системой и должен быть записан в файл куста на диске, а сброшенный бит означает, что его сектор не обновлялся.

При создании нового или изменении уже существующего раздела или параметра диспетчер конфигурации отмечает модифицированные секторы куста в массиве измененных секторов. Далее он планирует операцию отложенной записи, или *синхронизацию куста* (hive sync). Системный поток отложенной записи активизируется через 5 секунд после запроса на синхронизацию куста и записывает измененные секторы всех кустов из памяти на диск. Таким образом, система сбрасывает на диск и все изменения в данных реестра, произошедшие за период между запросом на синхронизацию и самой синхронизацией. Следующая синхронизация возможна не ранее, чем через 5 секунд.

> **ПРИМЕЧАНИЕ** В Windows Server 2003 можно изменить 5-секундную задержку по умолчанию, используемую системным потоком отложенной записи. Для этого модифицируйте в реестре параметр HKLM\System\CurrentControlSet\Session Manager\Configuration Manager\RegistryLazyFlushInterval.

Если бы поток отложенной записи сразу записывал все измененные секторы в файл куста и при этом произошел бы крах системы, то файл куста оказался бы в несогласованном состоянии, и возможность его восстановления была бы утрачена. Чтобы предотвратить такую ситуацию, массив измененных секторов куста и все измененные секторы сначала записываются в регистрационный куст, и при необходимости его размер увеличивается. Далее поток отложенной записи обновляет порядковый номер базового блока куста и записывает измененные секторы в файл. Закончив эту операцию, поток отложенной записи обновляет второй порядковый номер в базовом блоке. Поэтому, если в момент записи в куст система рухнет, при ее перезагрузке диспетчер конфигурации обнаружит, что два порядковых номера в базовом блоке куста не совпадают, и сможет обновить куст, используя измененные секторы из файла регистрационного куста (т. е. произведет операцию отката вперед). В результате данные куста останутся актуальными и в согласованном состоянии.

Для еще большей защиты целостности критически важного куста SYSTEM диспетчер конфигурации в Windows 2000 поддерживает на диске его зеркальную копию. Вы можете найти соответствующий файл в каталоге \Windows\System32\Config файл с именем System без атрибута «скрытый» — System.alt. Файл System.alt является резервной копией куста. При каждой синхронизации куста SYSTEM происходит обновление и System.alt. Если при запуске системы диспетчер конфигурации обнаружит повреждение куста SYSTEM, он попытается загрузить его резервную копию. Если она не повреждена, то будет использована для обновления исходного куста SYSTEM.

Windows XP и Windows Server 2003 не поддерживают куст System.alt, так как NTLDR в этих версиях Windows знает, как обрабатывать файл System.log для актуализации куста System, который пришел в рассогласованное состояние при выключении системы или ее крахе. В Windows Server 2003 внесены и другие усовершенствования для большей устойчивости к повреждениям реестра. До Windows Server 2003 диспетчер конфигурации вызывал крах системы, обнаружив базовый блок, приемник или ячейку с данными, которые не проходят проверки на целостность. Диспетчер конфигурации в Windows Server 2003 справляется с такими проблемами и, если повреждения не слишком сильны, заново инициализирует поврежденные структуры данных (с возможным удалением подразделов в ходе этого процесса) и продолжает работу. Если же ему нужно прибегнуть к самовосстановлению, он уведомляет об этом пользователя, отображая диалоговое окно с сообщением о системной ошибке.

ПРИМЕЧАНИЕ В каталоге \Windows\System32\Config также имеется скрытый файл System.sav. Это версия куста SYSTEM, которая служила изначальной копией куста System. Именно этот файл копируется программой Windows Setup с дистрибутива.

Оптимизация операций с реестром

Диспетчер конфигурации предпринимает некоторые меры для оптимизации операций с реестром. Во-первых, практически у каждого раздела реестра имеется дескриптор защиты от несанкционированного доступа. Но было бы очень неэффективно хранить копии уникальных дескрипторов защиты для каждого раздела в кусте, поскольку сходные параметры защиты часто применяются к целым ветвям дерева реестра. Когда система устанавливает защиту для какого-либо раздела, диспетчер конфигурации в Windows 2000 прежде всего проверяет дескриптор защиты его родительского раздела, а потом просматривает все подразделы родительского раздела. Если дескриптор защиты одного из них совпадает с дескриптором защиты, запрошенным системой для раздела, диспетчер конфигурации просто использует уже существующий дескриптор. Учет числа разделов, совместно использующих один и тот же дескриптор, ведется с помощью счетчика ссылок. В Windows XP и Windows Server 2003 диспетчер конфигурации проверяет пул уникаль-

ных дескрипторов защиты, чтобы убедиться, что в кусте имеется по крайней мере одна копия каждого уникального дескриптора защиты.

Диспетчер конфигурации также оптимизирует хранение имен разделов и параметров в кусте. Реестр полностью поддерживает Unicode, но, если в каком-либо имени присутствуют только ASCII-символы, диспетчер конфигурации сохраняет это имя в кусте в ASCII-формате. Когда диспетчер конфигурации считывает имя (например, при поиске по имени), он преобразует его формат в памяти в Unicode. Хранение имен в формате ASCII позволяет существенно уменьшить размер куста.

Чтобы свести к минимуму нагрузку на память, блоки управления разделами не хранят полные пути разделов реестра. Вместо этого они ссылаются лишь на имя раздела. Так, блок управления разделом \Registry\System\Control хранит не полный путь, а имя Control. Дополнительная оптимизация в использовании памяти выражается в том, что диспетчер конфигурации хранит имена разделов в блоках управления именами разделов (key name control blocks), и все блоки управления разделов с одним и тем же именем используют один и тот же блок управления именем раздела. Для ускорения просмотра диспетчер конфигурации хранит имена блоков управления разделами в специальной хэш-таблице.

Для быстрого доступа к блокам управления разделами диспетчер конфигурации сохраняет наиболее часто используемые блоки управления в кэш-таблице, сконфигурированной как хэш-таблица. При поиске блока диспетчер конфигурации первым делом проверяет кэш-таблицу. Более того, у диспетчера конфигурации имеется другой кэш, *таблица отложенного закрытия* (delayed close table), в которой хранятся блоки управления разделов, закрытых приложениями. В результате приложение при необходимости может быстро открыть недавно закрытый раздел. По мере добавления новых недавно закрытых блоков диспетчер удаляет из этой таблицы самые старые блоки.

Сервисы

Практически в каждой операционной системе есть механизм, запускающий при загрузке системы процессы, которые предоставляют сервисы, не увязываемые с интерактивным пользователем. В Windows такие процессы называются *сервисами*, или *Windows-сервисами*, поскольку при взаимодействии с системой они полагаются на Windows API. Сервисы аналогичны демонам UNIX и часто используются для реализации серверной части клиент-серверных приложений. Примером Windows-сервиса может служить Web-сервер, поскольку он должен запускаться вместе с системой и работать независимо от того, зарегистрировался ли в системе какой-нибудь пользователь.

Windows-сервисы состоят из трех компонентов — сервисного приложения (service application), программы управления сервисом (service control program, SCP) и диспетчера управления сервисами (service control manager, SCM). Для начала мы обсудим сервисные приложения, учетные записи сервисов и работу SCM. Далее мы поясним, как происходит автоматический за-

пуск сервисов при загрузке системы, и рассмотрим, что делает SCM в случае сбоя сервиса при его загрузке и как он завершает работу сервисов.

Сервисные приложения

Сервисные приложения вроде Web-серверов состоят минимум из одной программы, выполняемой как Windows-сервис. Для запуска, остановки и настройки сервиса предназначена SCP. Хотя в Windows имеются встроенные SCP, обеспечивающие базовую функциональность для запуска, остановки, приостановки и возобновления сервисных приложений, некоторые сервисные приложения предоставляют собственные SCP, позволяющие администраторам указывать параметры конфигурации того сервиса, которым они управляют.

Сервисные приложения представляют собой просто Windows-программы (GUI или консольные) с дополнительным кодом для обработки команд от SCM и возврата ему статусной информации. Поскольку у большинства сервисов нет пользовательского интерфейса, они создаются в виде консольных программ.

Когда вы устанавливаете приложение, в состав которого входит некий сервис, программа установки приложения должна зарегистрировать этот сервис в системе. Для его регистрации вызывается Windows-функция *CreateService*, реализованная в Advapi32.dll (\Windows\System32\Advapi32.dll). Эта DLL, название которой расшифровывается как «Advanced API», реализует всю клиентскую часть SCM API.

Регистрируя сервис через *CreateService*, программа установки посылает SCM сообщение о том, где будет находиться данный сервис. Затем SCM создает в реестре раздел для сервиса по адресу HKLM\SYSTEM\CurrentControlSet\Services. Раздел Services является постоянным представлением базы данных SCM. Индивидуальные разделы для каждого сервиса определяют пути к соответствующим исполняемым файлам, а также параметры и конфигурационные настройки сервисов.

Зарегистрировав сервис, программа установки или управляющее приложение может запустить его через функцию *StartService*. Поскольку некоторые приложения, основанные на сервисах, нужно инициализировать при загрузке системы, программа установки обычно регистрирует сервис как автоматически запускаемый, просит пользователя перезагрузить систему для завершения установки и при загрузке системы позволяет SCM запустить сервис.

При вызове *CreateService* программа должна указывать некоторые параметры, описывающие характеристики сервиса. Эти характеристики включают тип сервиса (выполняется ли этот сервис в собственном процессе или совместно с другими сервисами), местонахождение его исполняемого файла, необязательное экранное имя, необязательные имя и пароль для запуска сервиса в контексте защиты определенной учетной записи, тип запуска (запускается ли он автоматически при загрузке системы или вручную под управлением SCP), код, указывающий, как система должна реагировать на ошибку при запуске сервиса, и необязательную информацию о моменте запуска относительно других сервисов (если данный сервис запускается автоматически).

SCM хранит каждую характеристику как параметр в разделе, созданном для данного сервиса. Пример такого раздела реестра показан на рис. 4-8.

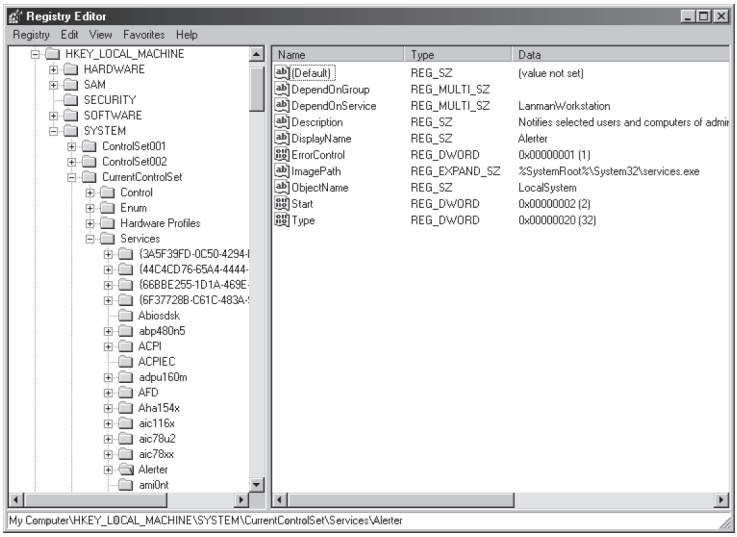


Рис. 4-8. *Пример раздела реестра для сервиса*

В таблице 4-7 перечислены характеристики сервиса, многие из которых применимы и к драйверам устройств. (Заметьте, что не все из них свойственны каждому типу сервисов или драйверов устройств.) Для хранения собственной конфигурационной информации сервиса в его разделе создается подраздел Parameters, в котором и будут находиться параметры конфигурации этого сервиса. Сервис получает значения параметров через стандартные функции реестра.

ПРИМЕЧАНИЕ SCM не обращается к подразделу Parameters сервиса до тех пор, пока данный сервис не удаляется; лишь в момент его удаления SCM уничтожает весь раздел сервиса вместе с подразделами вроде Parameters.

Таблица 4-7. *Параметры реестра для сервисов и драйверов*

Параметр	Значения	Описание
Start	SERVICE_BOOT_START (0)	Ntldr или Osloader предварительно загружают драйвер, чтобы во время загрузки он находился в памяти. Подобные драйверы инициализируются непосредственно перед драйверами «типа» SERVICE_SYSTEM_START
	SERVICE_SYSTEM_START (1)	Драйвер загружается и инициализируется после инициализации драйверов «типа» SERVICE_BOOT_START

см. след. стр.

Таблица 4-7. (*продолжение*)

Параметр	Значения	Описание
	SERVICE_AUTO_START (2)	Драйвер или сервис запускается SCM после запуска SCM-процесса Services.exe
	SERVICE_DEMAND_START (3)	Драйвер или сервис запускается SCM по требованию
	SERVICE_DISABLED (4)	Драйвер или сервис не загружается и не инициализируется
ErrorControl	SERVICE_ERROR_IGNORE (0)	Диспетчер ввода-вывода игнорирует возвращаемые драйвером или сервисом коды ошибок. Предупреждения не выводятся и не регистрируются
	SERVICE_ERROR_NORMAL (1)	Если драйвер или сервис сообщает об ошибке, выводится предупреждение
	SERVICE_ERROR_SEVERE (2)	Если драйвер или сервис сообщает об ошибке и последняя удачная конфигурация еще не используется, загружается именно эта конфигурация. Если она уже используется, загрузка продолжается
	SERVICE_ERROR_CRITICAL (3)	Если драйвер или сервис сообщает об ошибке и последняя удачная конфигурация еще не используется, загружается именно эта конфигурация. Если она уже используется, загрузка останавливается и выводится «синий экран»
Type	SERVICE_KERNEL_DRIVER (1)	Драйвер устройства
	SERVICE_FILE_SYSTEM_ DRIVER (2)	Драйвер файловой системы режима ядра
	SERVICE_ADAPTER (4)	Устаревшее значение
	SERVICE_RECOGNIZER_ DRIVER (8)	Драйвер, распознающий файловую систему
	SERVICE_WIN32_OWN_ PROCESS (16)	Сервис выполняется в собственном процессе, предназначенном только для него
	SERVICE_WIN32_SHARE_ PROCESS (32)	Сервис выполняется в процессе, используемом несколькими сервисами
	SERVICE_INTERACTIVE_ PROCESS (256)	Сервис может выводить на консоль окна и принимать ввод от пользователя
Group	Имя группы	Драйвер или сервис инициализируется при инициализации этой группы
Tag	Номер тэга	Указывает порядковый номер в последовательности инициализации группы. Этот параметр неприменим к сервисам
ImagePath	Путь к исполняемому файлу сервиса или драйвера	Если *ImagePath* не определен, диспетчер ввода-вывода ищет драйверы в \Windows\System32\Drivers, а SCM использует Windows-функции для поиска образа по значению переменной окружения PATH

Таблица 4-7. (*окончание*)

Параметр	Значения	Описание
DependOn Group	Имя группы	Драйвер или сервис не загружается, пока не будет загружен драйвер или сервис из указанной группы
Depend OnService	Имя сервиса	Сервис не загружается, пока не будет загружен указанный сервис. Этот параметр неприменим к драйверам устройств, кроме тех, у которых тип запуска определен как SERVICE_AUTO_START
ObjectName	Обычно Local System, но может быть именем учетной записи, например .\Administrator	Указывает учетную запись, под которой будет работать сервис. Если ObjectName не определен, используется учетная запись Local System. Этот параметр неприменим к драйверам устройств
DisplayName	Имя сервиса	Имя сервиса, показываемое сервисным приложением на экране. Если это имя не указано, именем сервиса считается имя его раздела в реестре
Description	Описание сервиса	Строка длиной до 32767 байтов
Failure Actions	Описание действий SCM при неожиданном завершении процесса сервиса	Действия при сбое включают перезапуск процесса сервиса, перезагрузку системы и запуск указанной программы. Этот параметр неприменим к драйверам
FailureCommand	Командная строка программы	SCM считывает этот параметр, только если в FailureActions указана необходимость запуска какой-либо программы при сбое сервиса. Этот параметр неприменим к драйверам
Security	Дескриптор защиты	Содержит дескриптор защиты, определяющий, кто и с какими правами может получать доступ к объекту сервиса, созданному SCM

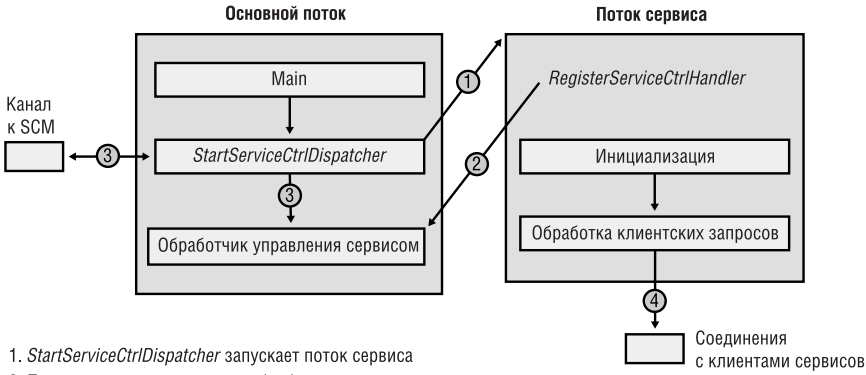
Заметьте, что к драйверам устройств применимы три значения параметра Type. Они используются драйверами устройств, которые также хранят свои параметры в разделе реестра Services. SCM отвечает за запуск драйверов со значением Start, равным SERVICE_AUTO_START или SERVICE_DE MAND_START, поэтому база данных SCM естественным образом включает и драйверы. Сервисы используют другие типы (SERVICE_WIN32_OWN_PROCESS и SERVICE_WIN32_SHARE_PROCESS), которые являются взаимоисключающими. Программы, содержащие более одного сервиса, указывают тип SERVICE_WIN32_SHARE_PROCESS. Преимущество совместного использования процесса несколькими сервисами — экономия ресурсов, которые в ином случае понадобились бы на диспетчеризацию индивидуальных процессов. Но потенциальный недостаток этой схемы в том, что работа всех сервисов данного процесса прекращается, если один из них вызывает ошибку, из-за которой за-

вершается их процесс. Кроме того, все сервисы в одном процессе должны выполняться под одной и той же учетной записью.

Когда SCM запускает сервисный процесс, тот немедленно вызывает *StartServiceCtrlDispatcher*. Эта функция принимает список точек входа в сервисы — по одной на каждый сервис процесса. Каждая точка входа идентифицируется именем соответствующего сервиса. Установив соединение с SCM по именованному каналу, *StartServiceCtrlDispatcher* входит в цикл, ожидая, когда от SCM поступит команда. SCM посылает команду запуска сервиса, и функция *StartServiceCtrlDispatcher* — всякий раз, когда получает такую команду, — создает поток, называемый *потоком сервиса* (service thread); он вызывает точку входа запускаемого сервиса и реализует цикл ожидания команд для сервиса. *StartServiceCtrlDispatcher* находится в бесконечном ожидании команд SCM и возвращает управление основной функции процесса только после остановки всех сервисов в этом процессе, давая ему время на очистку ресурсов.

Первое, что происходит при передаче управления входной точке сервиса, — вызов *RegisterServiceCtrlHandler*. Эта функция принимает и хранит указатель на функцию — обработчик управления (control handler), которую реализует сервис для обработки различных команд SCM. Она не взаимодействует с SCM, а лишь хранит только что упомянутую функцию в локальной памяти процесса для функции *StartServiceCtrlDispatcher*. Входная точка продолжает инициализацию сервиса, выделяя нужную память, создавая конечные точки коммуникационного канала и считывая из реестра данные о собственной конфигурации сервиса. По соглашению, которому следует большинство сервисов, эти параметры хранятся в подразделе Parameters раздела соответствующего сервиса. Входная точка, инициализируя сервис, может периодически посылать SCM сообщения о ходе процесса запуска сервиса (при этом используется функция *SetServiceStatus*). Когда входная точка заканчивает инициализацию, поток сервиса обычно входит в цикл ожидания запросов от клиентских приложений. Например, Web-сервер должен инициализировать TCP-сокет и ждать запросы на входящие HTTP-соединения.

Основной поток сервисного процесса, выполняемый в функции *StartServiceCtrlDispatcher*, принимает команды SCM, направляемые сервисам в этом процессе, и вызывает функцию — обработчик управления (хранимой *RegisterServiceCtrlHandler*) для соответствующего сервиса. SCM посылает следующие команды: stop (стоп), pause (пауза), resume (возобновление), interrogate (опрос), shutdown (выключение) и команды, определенные приложением. Схема внутренней организации сервисного процесса показана на рис. 4-9. На этой иллюстрации изображены два потока процесса, предоставляющего один сервис (основной поток и поток сервиса).



1. *StartServiceCtrlDispatcher* запускает поток сервиса
2. Поток сервиса регистрирует обработчик управления
3. *StartServiceCtrlDispatcher* вызывает обработчики в ответ на команды SCM
4. Поток сервиса обрабатывает клиентские запросы

Рис. 4-9. *Внутри процесса сервиса*

Утилита SrvAny

Если у вас есть какая-то программа, которую нужно запускать как сервис, вы должны модифицировать ее стартовый код в соответствии с требованиями к сервисам, кратко описанным в этом разделе. Если исходный код этой программы отсутствует, можно воспользоваться утилитой SrvAny из ресурсов Windows. SrvAny позволяет выполнять любое приложение как сервис. Она считывает путь файла, который должен быть загружен как сервис, из подраздела Parameters в разделе реестра, соответствующего данному сервису. При запуске SrvAny уведомляет SCM о том, что она предоставляет определенный сервис, и, получив от него команду, запускает исполняемый файл сервиса как дочерний процесс. Последний получает копию маркера доступа процесса SrvAny и ссылку на тот же объект WindowStation. Таким образом, сервис выполняется с параметрами защиты и настройками, указанными вами при конфигурировании SrvAny. Сервисы SrvAny не поддерживают значение параметра Type, соответствующее совместному использованию процесса, поэтому каждое приложение, установленное утилитой SrvAny как сервис, выполняется в отдельном процессе с отдельной копией хост-программы SrvAny.

Учетные записи сервисов

Контекст защиты сервиса очень важен для разработчиков и системных администраторов, поскольку он определяет, к каким ресурсам получит доступ этот сервис. Большинство сервисов выполняется в контексте защиты учетной записи локальной системы, если системным администратором или программой установки не указано иное. (В пользовательском интерфейсе название учетной записи локальной системы показывается как Local System или SYSTEM.) В Windows XP введены две разновидности учетной записи локаль-

ной системы: сетевой сервис (network service) и локальный сервис (local service). По сравнению с учетной записью локальной системы новые учетные записи обладают меньшими правами, и любой встроенный в Windows сервис, не требующий всей мощи учетной записи локальной системы, выполняется под соответствующей альтернативной учетной записью. Особенности этих учетных записей описываются в следующих разделах.

Учетная запись локальной системы

Под этой учетной записью выполняются базовые компоненты пользовательского режима, включая диспетчер сеансов (\Windows\System32\Smss.exe), процесс подсистемы Windows (Csrss.exe), подсистему локальной аутентификации (\Windows\System32\LSASS.exe) и процесс Winlogon (\Windows\System32\Winlogon.exe).

С точки зрения защиты, учетная запись Local System обладает исключительными возможностями — бóльшими, чем любая другая локальная или доменная учетная запись. Вот ее характеристики.

- Ее обладатель является членом группы локальных администраторов. В таблице 4-8 перечислены группы, которым назначается учетная запись локальной системы. (О том, как информация о членстве в группах используется при проверках прав доступа к объектам, см. в главе 8.)
- Она дает право на задание практически любых привилегий (даже таких, которые обычно не назначаются учетной записи локального администратора, например создания маркеров защиты). Список привилегий, назначаемых учетной записи Local System, приведен в таблице 4-9. (Описание каждой привилегии см. в главе 8.)
- Она дает право на полный доступ к большинству файлов и разделов реестра. Даже если какие-то объекты не разрешают полный доступ, процессы под этой учетной записью могут воспользоваться привилегией захвата объекта во владение (take-ownership privilege) и тем самым получить нужный вид доступа.
- Процессы, работающие под учетной записью Local System, применяют профиль пользователя по умолчанию (HKU\.DEFAULT). Поэтому им недоступна конфигурационная информация, которая хранится в профилях пользователей, сопоставленных с другими учетными записями.
- Если данная система входит в домен Windows, учетная запись Local System включает идентификатор защиты (SID) для компьютера, на котором выполняется сервисный процесс. Поэтому сервис, работающий под учетной записью Local System, будет автоматически аутентифицирован на других машинах в том же лесу. (*Лес* — это группа доменов в Active Directory.)
- Если только учетной записи компьютера специально не назначены права доступа (к общим сетевым ресурсам, именованным каналам и т. д.), процесс может получать доступ к сетевым ресурсам, разрешающим так называемые null-сеансы, т. е. соединения, не требующие соответствующих удостоверений защиты. Вы можете указывать общие ресурсы и каналы,

разрешающие null-сеансы на конкретном компьютере, в параметрах NullSessionPipes и NullSessionShares раздела реестра HKLM\SYSTEM\CurrentControlSet\Services\LanmanServer\Parameters.

Таблица 4-8. *Членство учетной записи сервиса в группах*

Local System	Network Service	Local Service
Everyone	Everyone	Everyone
Authenticated Users	Authenticated Users	Authenticated Users
Administrators	Users	Users
	Local	Local
	Network Service	Local Service
	Service	Service

Таблица 4-9. *Привилегии учетной записи сервиса*

Local System	Network Service	Local Service
SeAssignPrimaryTokenPrivilege	SeAssignPrimaryTokenPrivilege	SeAssignPrimaryTokenPrivilege
SeAuditPrivilege	SeAuditPrivilege	SeAuditPrivilege
SeBackupPrivilege	SeChangeNotifyPrivilege	SeChangeNotifyPrivilege
SeChangeNotifyPrivilege	SeIncreaseQuotaPrivilege	SeIncreaseQuotaPrivilege
SeCreateGlobalPrivilege	Привилегии, назначаемые группам Everyone, Authenticated Users и Users	Привилегии, назначаемые группам Everyone, Authenticated Users и Users
SeCreatePagefilePrivilege		
SeCreatePermanentPrivilege		
SeCreateTokenPrivilege[1]		
SeDebugPrivilege		
SeImpersonatePrivilege		
SeIncreaseBasePriorityPrivilege		
SeLoadDriverPrivilege		
SeLockMemoryPrivilege		
SeManageVolumePrivilege		
SeProfileSingleProcessPrivilege		
SeRestorePrivilege		
SeSecurityPrivilege		
SeShutdownPrivilege		
SeSystemEnvironmentPrivilege		
SeSystemTimePrivilege		
SeTakeOwnershipPrivilege		
SeTcbPrivilege		
SeUndockPrivilege		

[1] Учетная запись локальной системы в Windows Server 2003 эту привилегию не включает.

Учетная запись Network Service

Эта учетная запись предназначена для сервисов, которым нужно аутентифицироваться на других компьютерах в сети по учетной записи компьютера, как это делается в случае учетной записи Local System, но не требуется членство в административных группах или привилегии, назначаемые учетной записи Local System. Поскольку учетная запись Network Service не относится к группе администраторов, выполняемые под ней сервисы по умолчанию получают доступ к гораздо меньшему количеству разделов реестра, а также каталогов и файлов в файловой системе, чем учетная запись Local System. Более того, назначение меньшего числа привилегий ограничивает возможности скомпрометированного процесса сетевого сервиса. Например, процесс, работающий под учетной записью Network Service, не может загрузить драйвер устройства или открыть произвольный процесс.

Процессы, выполняемые под учетной записью Network Service, используют ее профиль; он загружается в раздел HKU\S-1-5-20, а его файлы и каталоги находятся в \Documents and Settings\NetworkService. В Windows XP и Windows Server 2003 под учетной записью Network Service выполняется DNS-клиент, отвечающий за разрешение DNS-имен и поиск контроллеров домена.

Учетная запись Local Service

Эта учетная запись практически идентична Network Service с тем отличием, что позволяет обращаться лишь к тем сетевым ресурсам, которые разрешают анонимный доступ. В таблице 4-9 показано, что у нее те же привилегии, что и у учетной записи Network Service, а таблица 4-8 — что она принадлежит к тем же группам (если не считать группы Network Service и Local Service). Профиль, используемый процессами, выполняемыми под учетной записью Local Service, загружается в HKU\S-1-5-19 и хранится в \Documents and Settings\LocalService.

В Windows XP и Windows Server 2003 под учетной записью Local Service работают такие компоненты, как Remote Registry Service (Служба удаленного реестра), предоставляющая удаленный доступ к реестру локальной системы, служба оповещения, принимающая широковещательные сообщения с административными уведомлениями, и служба LmHosts, обеспечивающая разрешение NetBIOS-имен.

Выполнение сервисов под другими учетными записями

В силу вышеописанных ограничений некоторые сервисы должны работать с удостоверениями защиты учетной записи пользователя. Вы можете сконфигурировать сервис на выполнение под другой учетной записью при его создании или с помощью оснастки Services (Службы) консоли MMC, указав в ней пароль и учетную запись, под которой должен работать сервис. В оснастке Services щелкните правой кнопкой мыши нужный сервис, выберите из контекстного меню команду Properties (Свойства), перейдите на вкладку Log On (Вход в систему) и щелкните переключатель This Account (С учетной записью), как показано на рис. 4-10.

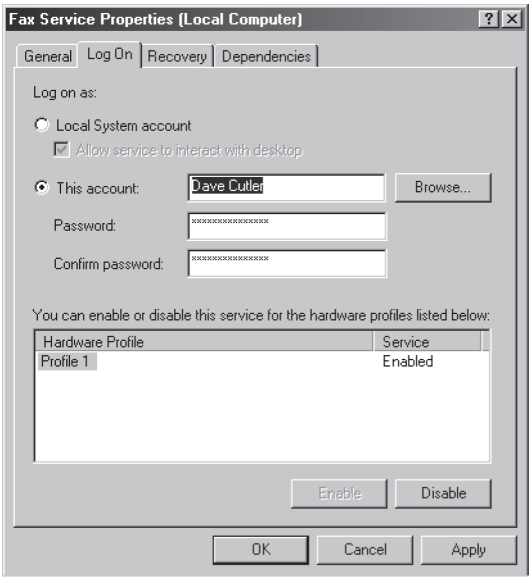


Рис. 4-10. *Параметры учетной записи сервиса*

Интерактивные сервисы

Другое ограничение сервисов, работающих под учетными записями Local System, Local Service или Network Service, заключается в том, что они не могут выводить окна на рабочий стол интерактивного пользователя (без специального флага в функции *MessageBox*, о котором мы расскажем чуть позже). Такое ограничение не является прямым следствием выполнения под этими учетными записями, а вызвано тем, как подсистема Windows назначает сервисные процессы объектам WindowStation.

Дело в том, что подсистема Windows сопоставляет каждый Windows-процесс с объектом WindowStation. Он включает объекты «рабочий стол», а те в свою очередь — объекты «окно». На консоли видим только объект WindowStation, и только он принимает пользовательский ввод от мыши и клавиатуры. В среде Terminal Services видимым является лишь один объект WindowStation для каждого сеанса, а все сервисы выполняются как часть консольного сеанса. Windows присваивает видимому объекту WindowStation имя WinSta0, и к нему обращаются все интерактивные процессы.

Если не указано иное, подсистема Windows сопоставляет сервисы, выполняемые под учетной записью Local System, с невидимым WindowStation-объектом Service-0x0-3e7$, который разделяется всеми неинтерактивными сервисами. Числовая часть его имени, 3e7, представляет назначаемый LSASS идентификатор сеанса входа в систему, который используется SCM для неинтерактивных сервисов, работающих под учетной записью Local System.

Сервисы, сконфигурированные на запуск под учетной записью пользователя (т. е. под учетной записью, отличной от Local System), выполняются в

другом невидимом объекте WindowStation, имя которого формируется из идентификатора, назначаемого LSASS сеансу входа в систему. На рис. 4-11 показано окно программы Winobj (ее можно скачать с сайта *www.sysinternals.com*) при просмотре каталога диспетчера объектов, в который Windows помещает объекты WindowStation. Обратите внимание на интерактивный WindowStation-объект WinSta0, неинтерактивный WindowStation-объект системного сервиса Service-0x0-3e7$ и неинтерактивный WindowStation-объект Service-0x0-6368f$, назначенный сервисному процессу, который зарегистрирован как пользователь.

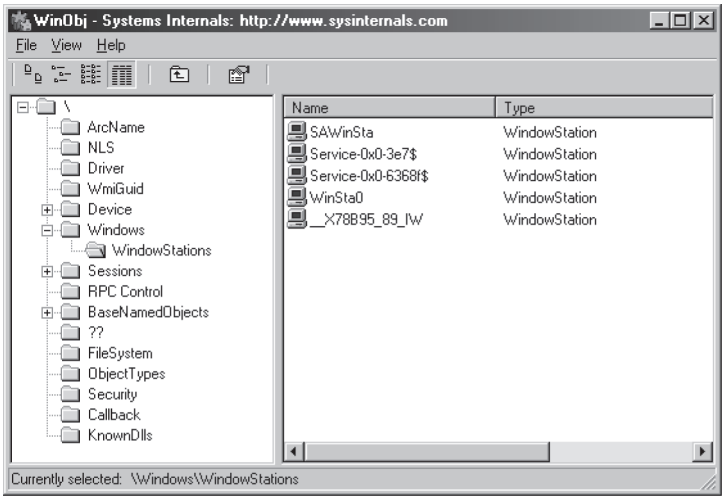


Рис. 4-11. *Список объектов WindowStation*

Независимо от того, работает ли сервис под учетной записью пользователя или под учетными записями Local System, Local Service либо Network Service, он не может получать пользовательский ввод или выводить окна на консоль, если он не сопоставлен с видимым объектом WindowStation. Фактически, если бы сервис попытался вывести обычное диалоговое окно, он бы казался зависшим, так как ни один пользователь не увидел бы это окно и не смог бы его закрыть с помощью мыши или клавиатуры. (Единственное исключение — вызов *MessageBox* со специальным флагом MB_SERVICE_NOTIFICATION или MB_DEFAULT_DESKTOP_ONLY. При MB_SERVICE_NOTIFICATION окно всегда выводится через интерактивный объект WindowStation, даже если сервис не сконфигурирован на взаимодействие с пользователем, а при MB_DEFAULT_DESKTOP_ONLY окно показывается на рабочем столе по умолчанию интерактивного объекта WindowStation.)

Иногда, хоть и очень редко, сервису нужно взаимодействовать с пользователем через информационные или диалоговые окна. Чтобы предоставить сервису право на взаимодействие с пользователем, в параметр Type в разделе реестра данного сервиса следует добавить модификатор SERVICE_INTERACTIVE_PROCESS. (Учтите, что сервисы, сконфигурированные для работы

под учетной записью пользователя, нельзя помечать как интерактивные.) В случае сервиса, помеченного как интерактивный, SCM запускает его процесс в контексте защиты учетной записи Local System, но сопоставляет сервис с WinSta0, а не с неинтерактивным объектом WindowStation. Это позволяет сервису выводить на консоль окна и реагировать на ввод пользователя.

ПРИМЕЧАНИЕ Microsoft не рекомендует запускать интерактивные сервисы (особенно под учетной записью Local System), так как это вредит безопасности. Windows, представленная интерактивным сервисом, уязвима перед оконными сообщениями, с помощью которых злонамеренный процесс, работающий как непривилегированный пользователь, может вызывать переполнения буферов в сервисном процессе и подменять его собой, чтобы повысить уровень своих привилегий в подсистеме защиты.

Диспетчер управления сервисами

Исполняемым файлом SCM является \Windows\System32\Services.exe, и подобно большинству процессов сервисов он выполняется как консольная Windows-программа. Процесс Winlogon запускает SCM на ранних этапах загрузки (см. главу 5) вызовом функции *SvcCtrlMain*. Она управляет запуском сервисов, сконфигурированных на автоматический старт. *SvcCtrlMain* выполняется почти сразу после появления на экране пустого рабочего стола, но, как правило, до загрузки процессом Winlogon графического интерфейса GINA, открывающего диалоговое окно для входа в систему.

Прежде всего *SvcCtrlMain* создает синхронизирующее событие с именем SvcCtrlEvent_A3752DX и в занятом состоянии. SCM освобождает этот объект только по завершении всех операций, необходимых для подготовки к получению команд от SCP. Последний устанавливает диалог с SCM через функцию *OpenSCManager*, которая не дает SCP связаться с SCM до его инициализации, реализуя это за счет ожидания перехода объекта SvcCtrlEvent_A3752DX в свободное состояние.

Далее *SvcCtrlMain* переходит к делу и вызывает функцию *ScCreateServiceDB*, создающую внутреннюю базу данных сервисов SCM. Функция *ScCreateServiceDB* считывает и сохраняет в разделе HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder\List параметр типа REG_MULTI_SZ, в котором содержится список имен и порядок определенных групп сервисов. Если сервису или драйверу нужно контролировать свой порядок запуска относительно сервисов других групп, в раздел реестра этого сервиса включается дополнительный параметр Group. Например, сетевой стек Windows построен по принципу «снизу вверх», поэтому сетевые сервисы должны указывать параметры Group, благодаря которым при загрузке системы они будут запускаться после загрузки сетевых драйверов. SCM создает в памяти внутренний список групп, где хранится порядок групп, считанный из реестра. В список входят NDIS, TDI, Primary Disk, Keyboard Port, Keyboard Class и другие группы. Дополнительные приложения и программное обеспечение от сторонних

разработчиков могут определять собственные группы и вносить их в список. Так, Microsoft Transaction Server добавляет группу MS Transactions.

ScCreateServiceDB сканирует раздел HKLM\SYSTEM\CurrentControlSet\Services и создает для каждого его подраздела запись в базе данных сервисов. Такая запись включает все параметры, определенные для сервиса, и поля, предназначенные для слежения за состоянием сервиса. SCM добавляет записи для драйверов устройств и сервисов, так как отвечает за запуск компонентов, помеченных для автоматического запуска. При этом SCM обнаруживает любые ошибки, вызываемые драйверами, которые помечены как запускаемые при загрузке системы (boot-start) и после нее (system-start). (Он также предоставляет приложениям средства для запроса состояния драйверов.) Диспетчер ввода-вывода загружает такие драйверы до начала выполнения какого-либо процесса пользовательского режима, поэтому они загружаются до старта SCM.

ScCreateServiceDB считывает параметр Group сервиса, определяя принадлежность этого сервиса к той или иной группе, и сопоставляет его значение с элементом в созданном ранее списке групп. Эта функция также считывает и сохраняет в базе данных зависимости сервиса от групп и других сервисов, запрашивая из реестра значения его параметров *DependOnGroup* и *DependOnService*. На рис. 4-12 показано, что представляет собой база данных SCM. Заметьте, что сервисы отсортированы в алфавитном порядке. Это вызвано тем, что SCM создает список на основе раздела Services, а Windows сортирует разделы реестра по алфавиту.

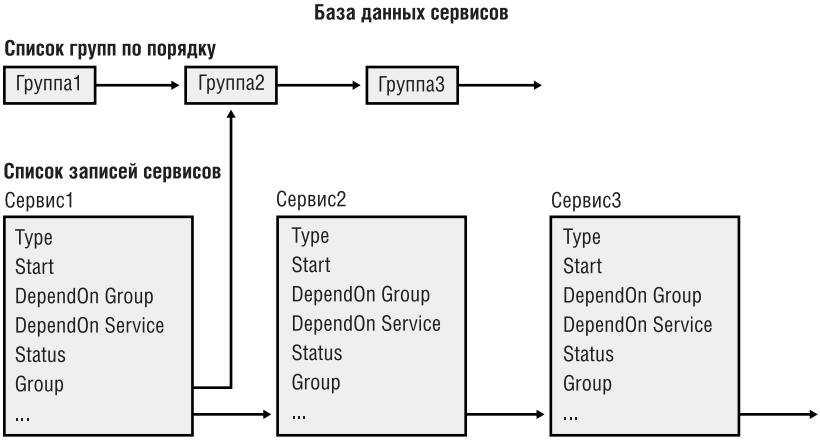


Рис. 4-12. *Организация базы данных сервисов*

При запуске сервиса SCM может понадобиться обращение к LSASS (например, для регистрации сервиса под учетной записью пользователя), поэтому он ждет, когда LSASS освободит синхронизирующее событие LSA_RPC_SERVER_ACTIVE, которое переходит в свободное состояние после инициализации LSASS. Winlogon тоже запускает процесс LSASS, поэтому инициализация LSASS

проходит параллельно инициализации SCM, но завершаться эти операции могут в разное время. После этого *SvcCtrlMain* вызывает *ScGetBootAndSystemDriverState*, которая сканирует базу данных сервисов и ищет записи для драйверов устройств, запускаемых при загрузке системы и после нее. *ScGetBootAndSystemDriverState* определяет, успешно ли запустился драйвер, проверяя наличие его имени в каталоге \Driver пространства имен диспетчера объектов. При успешной загрузке драйвера его объект помещается в данный каталог пространства имен диспетчером ввода-вывода, так что имена незагруженных драйверов присутствовать там не могут. Содержимое каталога Driver, полученное с помощью Winobj, показано на рис. 4-13. Если драйвер не загружен, SCM ищет его имя в списке, возвращаемом функцией *PnP_DeviceList*, которая сообщает о драйверах, включенных в текущий профиль оборудования системы. *SvcCtrlMain* отмечает имена драйверов из текущего профиля, которые не удалось запустить, в списке с именем ScFailedDrivers.

Перед запуском автоматически запускаемых сервисов SCM предпринимает еще несколько действий. Он создает именованный канал RPC с именем \Pipe\Ntsvcs, а затем RPC запускает поток, отслеживающий приходящие по этому каналу сообщения SCP. Далее SCM освобождает свой объект SvcCtrlEvent_A3752DX, сигнализируя о завершении инициализации.

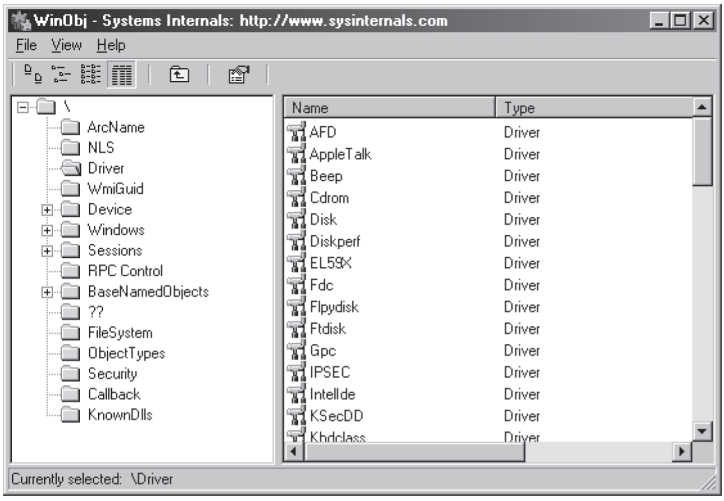


Рис. 4-13. *Список объектов драйверов*

Буквы сетевых дисков

SCM не только предоставляет интерфейс к сервисам, но и играет еще одну роль, никак не связанную с сервисами: он уведомляет GUI-приложения о создании или удалении сопоставления буквы с сетевым диском. SCM ждет, когда маршрутизатор многосетевого доступа (Multiple Provider Router, MPR) освободит объект \BaseNamedObjects\ScNetDrvMsg. Это происходит, когда приложение назначает букву диска удаленному

см. след. стр.

сетевому ресурсу или удаляет ее. (Сведения о MPR см. в главе 13.) При освобождении объекта-события маршрутизатором многосетевого доступа SCM вызывает Windows-функцию *GetDriveType*, чтобы получить список букв подключенных сетевых дисков. Если этот список изменяется в результате освобождения объекта-события, SCM посылает широковещательное Windows-сообщение типа WM_DEVICECHANGE с подтипом DBT_DEVICEREMOVECOMPLETE или DBT_DEVICEARRIVAL. Это сообщение адресовано главным образом Windows Explorer, чтобы он мог обновить любые открытые окна My Computer (Мой компьютер) с учетом изменений в наборе подключенных сетевых дисков.

Запуск сервиса

SvcCtrlMain вызывает SCM-функцию *ScAutoStartServices* для запуска всех сервисов, значение параметра Start которых указывает на автостарт. *ScAutoStartServices* также осуществляет автоматический запуск драйверов. Чтобы не запутаться, помните, что термином «сервисы» обозначаются как сервисы, так и драйверы, если явно не указано иное. Алгоритм *ScAutoStartServices* разбивает процесс запуска сервисов на несколько фаз, причем в каждой фазе запускаются определенные группы сервисов. Порядок запуска определяется параметром List в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder. Этот параметр, показанный на рис. 4-14, включает имена групп в том порядке, в каком они должны запускаться SCM. Таким образом, единственное, для чего сервис включается в ту или иную группу, — точная настройка момента его запуска относительно других сервисов.

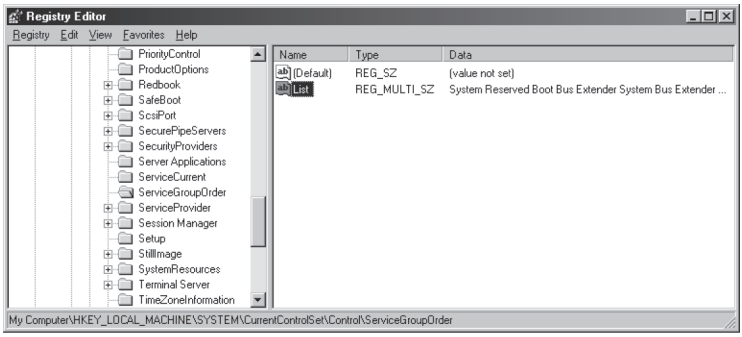


Рис. 4-14. *Раздел реестра ServiceGroupOrder*

В начале каждой фазы *ScAutoStartServices* отмечает все сервисы, относящиеся к группе, которая запускается на данной фазе. После этого *ScAutoStartServices* перебирает отмеченные сервисы, определяя возможность запуска каждого из них. При этом функция проверяет зависимость текущего сервиса от другой группы, на существование которой указывает наличие в соответствующем разделе реестра параметра DependOnGroup. Если сервис зависит от какой-либо группы, она должна быть уже инициализирована и

хотя бы один ее сервис должен быть успешно запущен. Если сервис зависит от группы, запускаемой позже группы данного сервиса, SCM генерирует ошибку, которая сообщает о «круговой зависимости». Если *ScAutoStartServices* имеет дело с Windows-сервисом или с автоматически запускаемым драйвером устройства, она дополнительно проверяет, зависит ли данный сервис от каких-либо других сервисов, и, если да, то запущены ли они. Зависимости сервисов указываются в параметре *DependOnService* раздела, соответствующего сервису. Если сервис зависит от других сервисов из групп, запускаемых позднее, SCM также генерирует ошибку, связанную с «круговой зависимостью». Если же сервис зависит от еще не запущенных сервисов из той же группы, он просто пропускается.

Если все зависимости корректны, *ScAutoStartServices* делает последнюю перед запуском сервиса проверку: входит ли он в состав загружаемой в данный момент конфигурации. В разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot имеется два подраздела — Minimal и Network. Какой из них будет использован SCM для проверки зависимостей, определяется типом безопасного режима, выбранного пользователем. Если пользователь выбрал Safe Mode (Безопасный режим) или Safe Mode With Command Prompt (Безопасный режим с поддержкой командной строки), SCM обращается к подразделу Minimal, а если пользователь выбрал Safe Mode With Networking (Безопасный режим с загрузкой сетевых драйверов), то — к подразделу Network. Наличие в разделе SafeBoot строкового параметра *Option* говорит не только о загрузке системы в безопасном режиме, но и указывает выбранный пользователем тип безопасного режима. Подробнее о безопасных режимах см. главу 5.

Как только SCM принимает решение о запуске сервиса, он вызывает *ScStartService*, которая запускает сервисы иначе, чем драйверы устройств. В случае Windows-сервиса эта функция сначала определяет имя файла, запускающего процесс сервиса, и для этого считывает параметр ImagePath из раздела, соответствующего сервису. Далее она определяет значение параметра Type, и, если оно равно SERVICE_WIN32_SHARE_PROCESS (0x20), SCM проверяет, зарегистрирован ли процесс, запускающий данный сервис, по той же учетной записи, что и запускаемый сервис. Учетная запись пользователя, под которой должен работать сервис, хранится в разделе этого сервиса в параметре ObjectName. Если параметр ObjectName сервиса содержит значение LocalSystem или этот параметр вовсе отсутствует, данный сервис запускается под учетной записью Local System.

SCM удостоверяется, что процесс сервиса еще не запущен под другой учетной записью. Для этого он ищет в своей внутренней базе данных, называемой *базой данных образов* (image database), запись для параметра ImagePath сервиса. Если такой записи нет, SCM создает ее. При этом он сохраняет имя учетной записи, используемой сервисом, и данные из параметра ImagePath. SCM требует от сервисов наличия параметра ImagePath. Если его нет, SCM генерирует ошибку, сообщая, что найти путь к сервису не удалось и запуск этого сервиса невозможен. Если SCM находит существующую запись в базе данных с теми же данными, что и в ImagePath, то проверяет, совпадают

ли сведения об учетной записи пользователя запускаемого сервиса с информацией в базе данных. Процесс регистрируется только под одной учетной записью, и поэтому SCM сообщает об ошибке, если имя учетной записи сервиса отличается от имени, указанного другим сервисом, который уже выполняется в том же процессе.

Чтобы зарегистрировать (если это указано в конфигурации сервиса) и запустить процесс сервиса, SCM вызывает *ScLogonAndStartImage*. SCM регистрирует сервисы, выполняемые не под системной учетной записью, с помощью LSASS-функции *LsaLogonUser*. Обычно *LsaLogonUser* требует пароль, но SCM указывает LSASS, что пароль хранится как «секрет» LSASS в разделе реестра HKLM\SECURITY\Policy\Secrets. (Учтите, что содержимое SECURITY обычно невидимо, поскольку его параметры защиты по умолчанию разрешают доступ только по системной учетной записи.) SCM, вызывая *LsaLogonUser*, указывает тип регистрации, и поэтому LSASS ищет пароль в подразделе раздела Secrets с именем типа _SC_*<имя сервиса>*. Конфигурируя регистрационную информацию сервиса, SCM командует LSASS сохранить регистрационный пароль как секрет, используя функцию *LsaStorePrivateData*. Если регистрация проходит успешно, *LsaLogonUser* возвращает описатель маркера доступа. В Windows такие маркеры применяются для установки контекста защиты пользователя, и SCM сопоставляет маркер доступа с процессом, реализующим сервис.

После успешной регистрации SCM загружает информацию из профиля учетной записи (если она еще не загружена), для чего вызывает функцию *LoadUserProfile* из \Windows\System32\Userenv.dll. Местонахождение куста, который *LoadUserProfile* загружает в реестр как раздел HKEY_CURRENT_USER, определяется параметром HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList*<раздел профиля пользователя>*\ProfileImagePath.

Интерактивный сервис должен открыть WindowStation-объект WinSta0. Но прежде чем разрешить интерактивному сервису доступ к WinSta0, *ScLogonAndStartImage* проверяет, установлен ли параметр HKLM\SYSTEM\CurrentControlSet\Control\Windows\NoInteractiveServices. Этот параметр устанавливается администратором для того, чтобы запретить сервисам, помеченным как интерактивные, вывод окон на консоль. Такой вариант применяется при работе сервера в необслуживаемом режиме, когда пользователей, взаимодействующих с интерактивными сервисами, нет.

Далее *ScLogonAndStartImage* запускает процесс сервиса, если он еще не выполняется. SCM запускает процессы в приостановленном состоянии, используя Windows-функцию *CreateProcessAsUser*. После этого SCM создает именованный канал для взаимодействия с процессом сервиса и присваивает ему имя \Pipe\Net\NtControlPipe*X*, где *X* — счетчик количества созданных SCM каналов. SCM возобновляет выполнение процесса сервиса через функцию *ResumeThread* и ждет подключения сервиса к созданному каналу. Сколько времени SCM ждет вызова сервисом функции *StartServiceCtrlDispatcher* и соединения с каналом, зависит от значения параметра реестра HKLM\SYSTEM\CurrentControlSet\Control\ServicesPipeTimeout (если такой параметр

существует). По истечении этого времени SCM завершает процесс и считает запуск сервиса несостоявшимся. Если параметра ServicesPipeTimeout в реестре нет, SCM использует таймаут по умолчанию, равный 30 секундам. Этот же таймаут распространяется на все виды коммуникационной связи между SCM и сервисами.

Когда сервис подключается по каналу к SCM, последний посылает сервису команду запуска. Если сервис в течение определенного периода не реагирует на нее, SCM переходит к запуску следующего сервиса. В данном случае SCM не завершает процесс сервиса, а заносит запись об ошибке в системный журнал событий, указывая, что сервис не смог своевременно начать работу.

Если параметр реестра Type для сервиса, запускаемого SCM с помощью *ScStartService*, равен SERVICE_KERNEL_DRIVER или SERVICE_FILE_SYSTEM_DRIVER, то данный сервис является драйвером устройства, и *ScStartService* вызывает для его загрузки *ScLoadDeviceDriver*. Она выдает процессу SCM право на загрузку драйвера и вызывает сервис ядра *NtLoadDriver*, передавая ему значение параметра реестра ImagePath для данного драйвера. В отличие от сервисов драйверам не обязательно указывать значение ImagePath. Если оно не указано, SCM формирует путь к образу исполняемого файла, добавляя имя драйвера к строке \Windows\System32\Drivers\.

ScAutoStartServices продолжает поочередно обрабатывать сервисы, принадлежащие какой-либо группе, до тех пор, пока все они не будут запущены или не вызовут ошибку, связанную с нарушением зависимостей. Такая циклическая обработка позволяет SCM автоматически упорядочивать сервисы внутри группы в соответствии с их параметрами DependOnService. SCM сначала запускает сервисы, на которые полагаются другие сервисы, пропуская зависимые сервисы. Заметьте, что SCM игнорирует параметры Tag в подразделах Windows-сервисов раздела HKLM\SYSTEM\CurrentControlSet\Services. Эти параметры использует диспетчер ввода-вывода, упорядочивая запуск драйверов устройств в группе драйверов, запускаемых при загрузке и при старте системы.

Как только SCM завершает операции запуска для всех групп, перечисленных в списке ServiceGroupOrder\List, он переходит к запуску сервисов, принадлежащих к группам, которые не входят в этот список, а потом обрабатывает сервисы, не включенные ни в одну группу. Закончив, SCM переводит событие \BaseNamedObjects\SC_AutoStartComplete в свободное состояние.

Ошибки при запуске

Если драйвер или сервис в ответ на команду запуска SCM сообщает об ошибке, реакция SCM определяется значением параметра ErrorControl из раздела реестра для соответствующего сервиса. Если ErrorControl равен SERVICE_ERROR_IGNORE (0) или вообще не указан, SCM игнорирует ошибку и продолжает обработку запуска сервисов. Если ErrorControl равен SERVICE_ERROR_NORMAL (1), SCM заносит в журнал событий запись такого вида: «The <*имя сервиса*> service failed to start due to the following error:» («Служба <*имя сервиса*> завершена из-за ошибки:»). SCM добавляет возвращаемый сервисом

Windows-код ошибки, указывая его в записи в качестве причины сбоя при запуске. На рис. 4-15 показан пример такой записи.

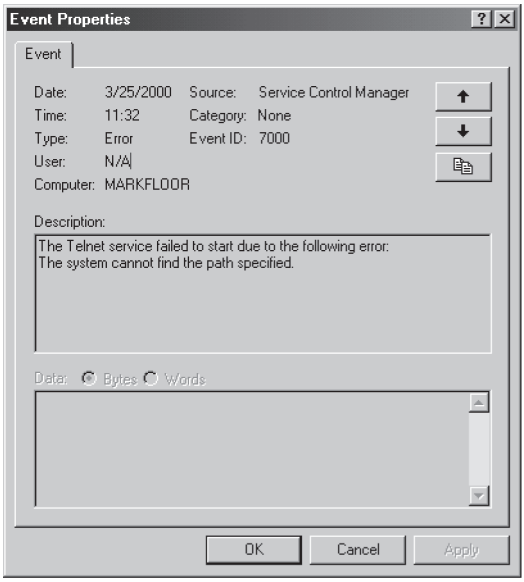


Рис. 4-15. *Запись в журнале событий, уведомляющая об ошибке при запуске сервиса*

Если сервис, значение ErrorControl которого равно SERVICE_ERROR_SEVERE (2) или SERVICE_ERROR_CRITICAL (3), сообщает об ошибке при запуске, SCM делает запись в журнале событий и вызывает внутреннюю функцию *ScRevertToLastKnownGood*. Эта функция активизирует версию реестра, соответствующую последней удачной конфигурации, в которой система была успешно загружена. После этого она перезагружает систему, вызывая сервис *NtShutdownSystem*, реализуемый исполнительной системой. Если система уже загружена в последней удачной конфигурации, она просто перезагружается.

Критерии успешной загрузки и последняя удачная конфигурация

Кроме запуска сервисов система возлагает на SCM определение того, когда следует сохранять раздел реестра HKLM\SYSTEM\CurrentControlSet в качестве последней удачной конфигурации. Раздел Services входит в раздел CurrentControlSet, поэтому CurrentControlSet включает представление реестра из базы данных SCM. CurrentControlSet также включает раздел Control, где хранятся многие параметры конфигурации подсистем режима ядра и пользовательского режима. По умолчанию загрузка считается успешной, если были запущены все автоматически запускаемые драйверы и пользователь смог войти в систему. Загрузка считается неудачной, если система остановилась из-за краха драйвера устройства или если автоматически запускаемый сер-

вис с параметром ErrorControl, равным SERVICE_ERROR_SEVERE или SERVICE_ERROR_CRITICAL, сообщил об ошибке при запуске.

SCM узнает, успешно ли стартовали автоматически запускаемые сервисы, но уведомление об успешном входе пользователя он должен получить от Winlogon (\Windows\System32\Winlogon.exe). При входе пользователя Winlogon вызывает функцию *NotifyBootConfigStatus*, которая посылает сообщение SCM. После успешного старта автоматически запускаемых сервисов или приема сообщения от *NotifyBootConfigStatus* (в зависимости от того, какое из этих событий будет последним) SCM вызывает системную функцию *NtInitializeRegistry* для сохранения текущей конфигурации системы.

Сторонние разработчики программного обеспечения могут заменить определение успешного входа Winlogon собственным. Например, если в системе работает Microsoft SQL Server, загрузка считается успешной только после того, как SQL Server может принимать и обрабатывать транзакции. Разработчики указывают свое определение успешной загрузки, используя программу верификации загрузки и регистрируя ее местонахождение в параметре, который сохраняется в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\BootVerificationProgram. Кроме того, при установке программы верификации загрузки нужно отключить вызов Winlogon функции *NotifyBootConfigStatus*, присвоив параметру HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\ReportBootOk значение, равное 0. Если такая программа установлена, SCM запускает ее, закончив обработку автоматически запускаемых сервисов. Перед сохранением последней удачной конфигурации SCM ждет вызова *NotifyBootConfigStatus* из этой программы.

Windows поддерживает несколько копий CurrentControlSet, который на самом деле представляет собой символьную ссылку на одну из этих копий. Им присваиваются имена в виде HKLM\SYSTEM\ControlSet*nnn*, где *nnn* — порядковый номер вроде 001 или 002. Раздел HKLM\SYSTEM\Select хранит параметры, определяющие роль каждой копии CurrentControlSet. Так, если CurrentControlSet ссылается на ControlSet001, значение параметра Current в разделе Select равно 1. Параметр LastKnownGood в разделе Select хранит номер последней удачной конфигурации. В разделе Select может быть еще один параметр, Failed, указывающий номер конфигурации, загрузка которой признана неудачной и прервана, после чего была предпринята попытка использования последней удачной конфигурации. На рис. 4-16 показаны наборы CurrentControlSet и параметры раздела Select.

NtInitializeRegistry синхронизирует набор параметров последней удачной конфигурации с содержимым дерева CurrentControlSet. После первой успешной загрузки системы последней удачной конфигурации еще нет, и система создает для нее новый набор параметров. Если же набор параметров последней удачной конфигурации уже есть, система просто обновляет его данные так, чтобы они совпадали с данными из CurrentControlSet.

Последняя удачная конфигурация может помочь, когда изменения, внесенные в CurrentControlSet при оптимизации системы или установке сервиса, вызывают сбои при следующей загрузке. Нажав клавишу F8 в самом на-

чале загрузки, пользователь может вызвать меню, которое позволит ему активизировать последнюю удачную конфигурацию и вернуть реестр к исходному состоянию (детали см. в главе 5).

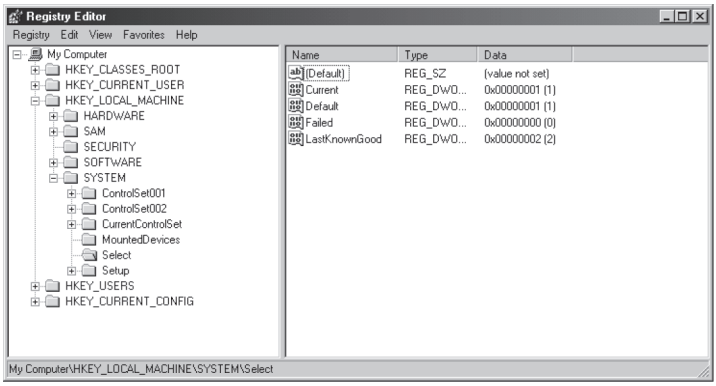


Рис. 4-16. *Раздел, предназначенный для выбора конфигурации*

Сбои сервисов

В разделах сервисов могут присутствовать необязательные параметры *FailureActions* и *FailureCommand*, записываемые SCM при запуске сервисов. Система уведомляет SCM о неожиданном завершении процесса сервиса, так как SCM соответствующим образом регистрируется в системе. В этом случае SCM определяет, какие сервисы выполнялись в этом процессе, и предпринимает действия по их восстановлению. Эти действия определяются параметрами реестра, относящимися к сбоям сервисов.

Сервисы могут указывать для SCM такие действия, как перезапуск сервиса, запуск какой-либо программы и перезагрузка компьютера. Более того, сервис может задавать действия, предпринимаемые при первом, втором и последующих сбоях его процесса, а также задавать период ожидания SCM перед перезапуском сервиса, если это действие определено сервисом. Например, сбой IIS Admin Service заставляет SCM запустить приложение IISReset, которое освобождает ресурсы и перезапускает сервис. Вы можете легко настроить действия, предпринимаемые для восстановления сервиса, на вкладке Recovery (Восстановление) окна свойств сервиса в оснастке Services (Службы), как показано на рис. 4-17.

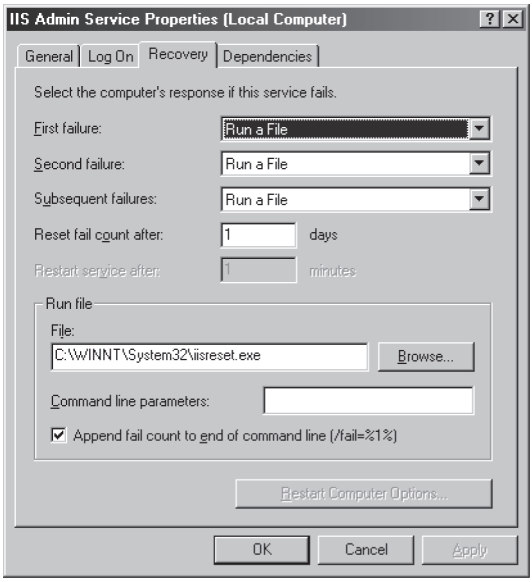


Рис. 4-17. *Параметры восстановления сервиса*

Завершение работы сервиса

Когда Winlogon вызывает Windows-функцию *ExitWindowsEx*, она посылает сообщение Csrss, процессу подсистемы Windows, для вызова его процедуры завершения. Csrss по очереди уведомляет активные процессы о завершении работы системы. Для каждого процесса, кроме SCM, Csrss ждет его завершения в течение времени, указанного в HKU\.DEFAULT\Control Panel\Desktop\ WaitToKillAppTimeout (по умолчанию — 20 секунд). Дойдя до SCM, Csrss также уведомляет его о завершении работы системы, но использует особый таймаут. Csrss опознает SCM по идентификатору процесса, сохраненному SCM при инициализации системы вызовом *RegisterServicesProcess*. Таймаут SCM отличен от таймаутов других процессов из-за того, что он обменивается сообщениями с сервисами. При завершении работы сервисы должны освободить ресурсы, поэтому администратору может быть достаточно настроить лишь таймаут SCM. Это значение хранится в HKLM\SYSTEM\CurrentControlSet\ Control\WaitToKillServiceTimeout и по умолчанию равно 20 секундам.

Обработчик завершения SCM отвечает за рассылку уведомлений о завершении работы всем сервисам, которые сообщали при инициализации о необходимости посылки им таких уведомлений. SCM-функция *ScShutdownAllServices* ищет в базе данных SCM сервисы, которым требуется уведомление о завершении работы, и посылает каждому из них команду на завершение. Для каждого сервиса, которому посылается уведомление о завершении работы, SCM фиксирует срок ожидания (wait hint), который указывается и самим сервисом при его регистрации. SCM определяет наибольший срок ожидания. Разослав уведомления о завершении работы, SCM ждет, пока не завер-

шится хотя бы один из получивших уведомление сервисов или пока не истечет наибольший срок ожидания.

Если по истечении этого срока сервис так и не завершился, SCM проверяет, не получил ли он сообщения о ходе процесса завершения от одного или нескольких ожидаемых им сервисов. Если хотя бы один сервис прислал такое сообщение, SCM снова ждет в течение периода, равного сроку ожидания. SCM выходит из этого цикла ожидания, если все сервисы завершились или если ни один из них не прислал ему сообщение о ходе процесса завершения.

Пока SCM занят рассылкой уведомлений и ожиданием завершения сервисов, Csrss ожидает завершения SCM. Если период ожидания Csrss (равный значению WaitToKillServiceTimeout) истекает, а SCM все еще выполняется, Csrss просто переходит к другим операциям, необходимым для завершения работы системы. Таким образом, сервисы, не прекратившие свою работу вовремя, продолжают выполняться вместе с SCM до момента выключения системы. К сожалению, нет никакого способа, который позволил бы администратору узнать, надо ли увеличить значение WaitToKillServiceTimeout, чтобы все сервисы успевали завершаться до выключения системы. (Подробнее о процессе выключения системы см. в главе 5.)

Разделяемые процессы сервисов

Выполнение каждого сервиса в собственном процессе может оказаться очень расточительным по отношению к системным ресурсам. С другой стороны, если в каком-то из сервисов, совместно использующих один процесс, возникает ошибка, вызывающая завершение этого процесса, работа всех сервисов этого процесса прекращается.

Некоторые из встроенных сервисов Windows выполняются в собственных процессах, а некоторые делят процессы совместно с другими сервисами. Например, процесс SCM включает сервисы Event Log (Журнал системы) и Plug and Play пользовательского режима, а процесс LSASS содержит такие службы защиты, как Security Accounts Manager (SamSs) (Диспетчер учетных записей безопасности), Net Logon (Netlogon) (Сетевой вход в систему) и IPSec Policy Agent (PolicyAgent) (Агент политики IP-безопасности).

Существует также «универсальный» процесс, Service Host (SvcHost) (\Windows\System32\Svchost.exe), который включает множество разнообразных сервисов. В различных процессах может выполняться несколько экземпляров SvcHost. К сервисам, размещаемым в процессах SvcHost, относятся, в частности, Telephony (TapiSrv), Remote Procedure Call (RpcSs) и Remote Access Connection Manager (RasMan). Windows реализует сервисы, выполняемые в SvcHost, в виде DLL и включает в раздел реестра каждого из этих сервисов определение ImagePath в формате «%SystemRoot%\System32\Svchost.exe –k netsvcs». В подразделе Parameters раздела такого сервиса должен присутствовать и параметр *ServiceDll*, указывающий на DLL-файл сервиса.

Для всех сервисов, использующих общий процесс SvcHost, должен быть указан один и тот же параметр («–k netsvcs» — в примере из предыдущего абзаца), чтобы у них была одна запись в базе данных образов. Когда SCM,

запуская сервисы, обнаруживает первый сервис с ImagePath, указывающим на SvcHost с каким-то параметром, он создает новую запись в базе данных образов и запускает процесс SvcHost с этим параметром. Новый процесс SvcHost принимает этот параметр и ищет одноименный параметр в разделе реестра HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost. SvcHost считывает его, интерпретируя как список имен сервисов, и при регистрации уведомляет SCM о предоставляемых сервисах. Пример раздела Svchost показан на рис. 4-18. Здесь процесс Svchost, запущенный с параметром «–k netsvcs», настроен на выполнение нескольких сетевых сервисов.

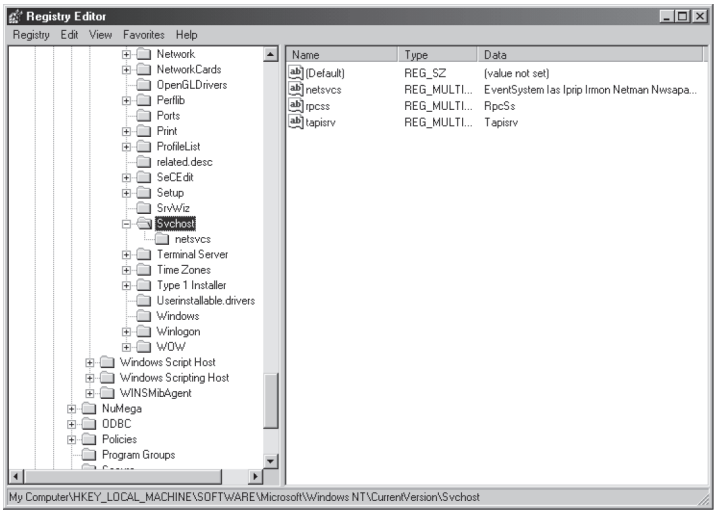


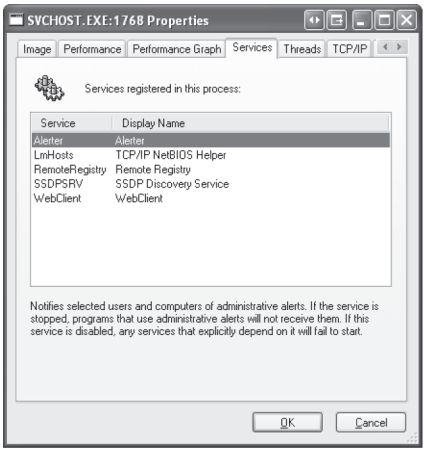
Рис. 4-18. *Раздел реестра Svchost*

Если при запуске сервисов SCM обнаруживает сервис SvcHost со значением ImagePath, которое соответствует уже существующей записи в базе данных образов, он не запускает второй процесс, а просто посылает уже выполняемому SvcHost команду на запуск этого сервиса. Существующий процесс SvcHost считывает из раздела реестра сервиса параметр *ServiceDll* и загружает DLL этого сервиса.

ЭКСПЕРИМЕНТ: просмотр сервисов, выполняемых в процессах

Утилита Process Explorer (*www.sysinternals.com*) выводит детальные сведения о сервисах, выполняемых внутри процессов. Запустите Process Explorer и откройте вкладки Services в окнах свойств для следующих процессов: Services.exe, Lsass.exe и Svchost.exe. В вашей системе должно выполняться несколько экземпляров SvcHost, и вы сможете увидеть, под какой учетной записью работает каждый из них, добавив столбец Username в окне Process Explorer или взглянув на поле Username на вкладке Image окна свойств процесса. На иллюстрации ниже показан список сервисов, выполняемых внутри SvcHost, который работает под учетной записью локальной системы.

см. след. стр.



Отображаемая информация включает имя сервиса, его экранное имя (display name) и описание (если таковое есть); описание выводится в Process Explorer внизу списка при выборе конкретного сервиса.

Просмотреть список сервисов, выполняемых внутри процессов, также можно с помощью утилит командной строки tlist.exe из Windows Support Tools и Tasklist, которая входит в состав Windows XP и Windows Server 2003. В первом случае синтаксис для просмотра сервисов выглядит так:

```
tlist /s
```

А во втором — так:

```
tasklist /svc
```

Заметьте, что эти утилиты не показывают описания или экранные имена сервисов.

Программы управления сервисами

Программы управления сервисами (service control programs, SCP) — стандартные Windows-приложения, использующие SCM-функции управления сервисами *CreateService*, *OpenService*, *StartService*, *ControlService*, *QueryServiceStatus* и *DeleteService*. Для вызова SCM-функций SCP сначала должна установить канал связи с SCM через функцию *OpenSCManager*. При запросе на открытие канала связи SCP должна указать, какие действия ей нужно выполнить. Например, если SCP требуется просто вывести список сервисов, присутствующих в базе данных SCM, то при вызове она запрашивает доступ для перечисления сервисов (enumerate-service access). При инициализации SCM создает внутренний объект, представляющий его базу данных. Для защиты этого объекта применяются функции защиты Windows. В частности, для него создается дескриптор защиты, определяющий, по каким учетным записям можно открывать объект и с какими правами. Например, в дескрипторе защиты

указывается, что получить доступ к объекту SCM для перечисления сервисов может группа Authenticated Users, но открыть объект SCM для создания или удаления сервиса могут только администраторы.

SCM реализует защиту не только своей базы данных, но и сервисов. Создавая сервис вызовом *CreateService*, SCP указывает дескриптор защиты, сопоставляемый с записью сервиса в базе данных. SCM хранит дескриптор защиты в параметре *Security* раздела, соответствующего сервису. При инициализации SCM сканирует раздел Services и считывает дескриптор защиты, так что параметры защиты сохраняются между загрузками системы. SCP должна указывать SCM тип доступа к сервису при вызове *OpenService* по аналогии с тем, как она это делает, вызывая *OpenSCManager* для обращения к базе данных SCM. SCP может запрашивать доступ для получения информации о состоянии сервиса, а также для его настройки, остановки и запуска.

SCP, которая вам, наверное, хорошо знакома, — оснастка Services (Службы) консоли MMC в Windows. Эта оснастка содержится в файле Windows\System32\Filemgmt.dll. Windows XP и Windows Server 2003 включают SCP командной строки Sc.exe (Service Controller), а для Windows 2000 такая программа доступна в ресурсах Windows 2000.

Иногда SCP расширяет политику управления сервисами по сравнению с той, которая реализуется SCM. Удачный пример — таймаут, устанавливаемый оснасткой Services при запуске сервисов (служб) вручную. Эта оснастка выводит индикатор, отражающий ход запуска сервиса. Если SCM ждет ответа сервиса на команду запуска неопределенно долго, то оснастка Services — только 2 минуты, после чего сообщает, что сервис не смог своевременно начать работу. Сервисы косвенно взаимодействуют с SCP-программами, изменяя свой статус, который отражает их прогресс в выполнении команды SCM. SCP запрашивают этот статус через функцию *QueryServiceStatus* и способны отличать зависшие сервисы от активно обновляющих свой статус. Благодаря этому они могут уведомлять пользователя о том, что делает тот или иной сервис.

Windows Management Instrumentation

В Windows NT всегда были интегрированные средства мониторинга производительности и системных событий. Приложения и система, как правило, используют Event Manager (Диспетчер событий) для вывода сообщений об ошибках и диагностической информации. С помощью Event Viewer (Просмотр событий) администраторы могут просматривать список событий как на локальном компьютере, так и на любом компьютере в сети. Аналогичным образом механизм поддержки счетчиков производительности позволяет приложениям и операционной системе передавать соответствующие статистические сведения таким программам мониторинга производительности, как Performance Monitor.

Хотя средства мониторинга событий и производительности в Windows NT 4 решают поставленные при разработке задачи, для них характерны

некоторые ограничения. Так, их программные интерфейсы различаются, что усложняет приложения, использующие для сбора данных не только мониторинг событий, но и мониторинг производительности. Вероятно, самый серьезный недостаток средств мониторинга в Windows NT 4 в том, что они слабо расширяемы (если вообще расширяемы) и не поддерживают двустороннее взаимодействие, необходимое для API управления. Приложения должны предоставлять данные в жестко предопределенных форматах. Performance API не позволяет приложениям получать уведомления о событиях, связанных с производительностью, а приложения, запрашивающие уведомления о событиях Event Manager, не могут ограничиться конкретными типами событий или источниками. Наконец, клиенты любого из механизмов мониторинга не могут взаимодействовать через Event Manager или Performance API с провайдерами данных, относящихся к событиям или счетчикам производительности.

Для устранения этих ограничений и поддержки средств управления другими типами источников данных в Windows включен новый механизм управления, Windows Management Instrumentation (WMI) (Инструментарий управления Windows). WMI — это реализация Web-Based Enterprise Management (WBEM) (Управление предприятием на основе Web), стандарта, введенного промышленным консорциумом Distributed Management Task Force (DMTF). Стандарт WBEM определяет правила проектирования средств управления и сбора данных в масштабах предприятия, обладающих достаточной расширяемостью и гибкостью для управления локальными и удаленными системами, которые состоят из произвольных компонентов. Поддержка WMI добавляется в Windows NT 4 установкой Service Pack 4. WMI также поддерживается в Windows 95 OSR2, Windows 98 и Windows Millennium. Хотя многие сведения из этого раздела применимы ко всем платформам Windows, поддерживающим WMI, особенности реализации WMI, о которых мы здесь рассказываем, все же специфичны для Windows 2000, Windows XP и Windows Server 2003.

Архитектура WMI

WMI состоит из четырех главных компонентов, как показано на рис. 4-19: управляющих приложений, инфраструктуры WMI, компонентов доступа (провайдеров) и управляемых объектов. Первые являются Windows-приложениями, получающими сведения об управляемых объектах для последующей обработки или вывода. Пример простого управляющего приложения — утилита Performance (Производительность), использующая WMI вместо Performance API. К более сложным программам относятся промышленные средства управления, позволяющие администраторам автоматизировать инвентаризацию программно-аппаратных конфигураций всех компьютеров на своих предприятиях.

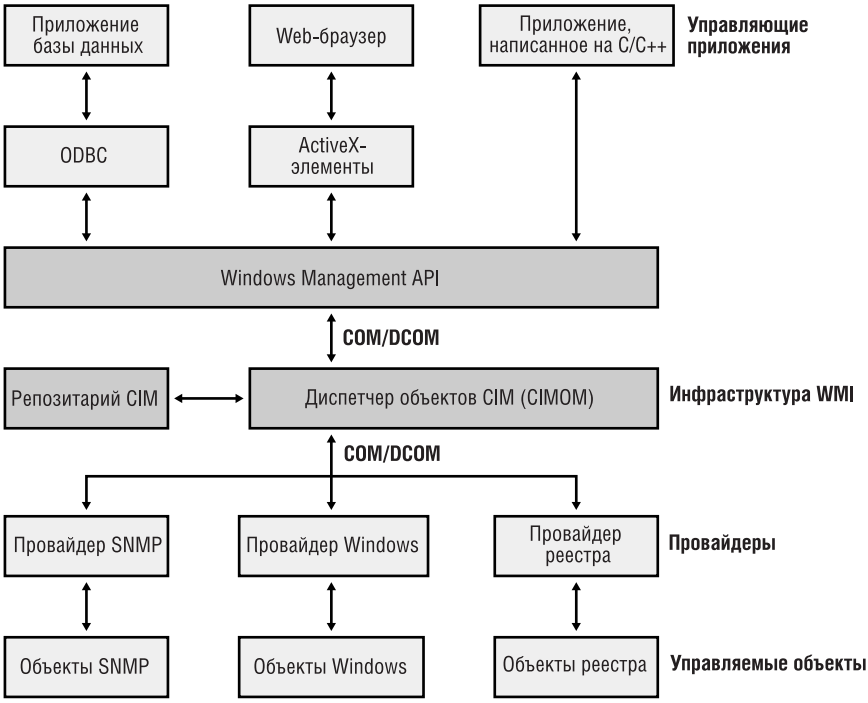


Рис. 4-19. *Архитектура WMI*

Управляющие приложения, как правило, предназначены для управления определенными объектами и сбора данных от них. Объект может представлять единственный компонент, например сетевую плату, или совокупность компонентов вроде компьютера. (Объект «компьютер» может включать объект «сетевая плата».) Провайдеры должны определять и экспортировать форму представления объектов, нужных управляющим приложениям. Так, изготовитель может добавить специфичную функциональность для сетевой платы, поддерживаемой WMI. Поэтому он должен написать свой провайдер, который открывал бы управляющим приложениям объекты, связанные с этой функциональностью.

Инфраструктура WMI, центральное место в которой занимает Common Information Model (CIM) Object Manager (CIMOM), связывает воедино управляющие приложения и провайдеры (о CIM — чуть позже). Инфраструктура также служит хранилищем классов объектов и зачастую диспетчером хранения свойств постоянных объектов. WMI реализует это хранилище в виде базы данных на диске, которая называется репозитарием объектов CIMOM (CIMOM Object Repository). WMI поддерживает несколько API, через которые управляющие приложения обращаются к данным объектов, провайдерам и определениям классов.

Для прямого взаимодействия с WMI Windows-программы используют WMI COM API, основной API управления. Остальные API размещаются поверх COM

API и включают ODBC-адаптер для СУБД Microsoft Access. Разработчик может использовать этот адаптер для встраивания ссылок на данные объектов в свою базу данных. После этого можно легко генерировать отчеты с помощью запросов к базе данных, содержащей WMI-информацию. ActiveX-элементы WMI поддерживают другой многоуровневый API. Web-разработчики используют ActiveX-элементы для создания Web-интерфейсов к WMI-данным. API для написания сценариев WMI представляет собой еще один API управления, используемый в приложениях, построенных на основе сценариев («скриптов»), и в программах Microsoft Visual Basic. Поддержка сценариев WMI предусмотрена во всех технологиях языков программирования Microsoft.

Как и для управляющих приложений, интерфейсы WMI COM образуют основной API для провайдеров. Но в отличие от управляющих приложений, являющихся COM-клиентами, провайдеры — это COM- или DCOM-серверы, т. е. провайдеры реализуют COM-объекты, с которыми взаимодействует WMI. Провайдеры могут быть реализованы в виде DLL, загружаемых в процесс диспетчера WMI, а также в виде отдельных Windows-приложений или сервисов. Microsoft предлагает ряд встроенных провайдеров, которые представляют данные из таких общеизвестных источников, как Performance API, реестр, Event Manager, Active Directory, SNMP и WDM-драйверы устройств. Сторонние разработчики могут создавать свои компоненты доступа, используя WMI SDK.

Провайдеры

В основе WBEM лежит спецификация CIM, разработанная DMTF. CIM определяет, как управляющие системы представляют любые компоненты вычислительной системы — от компьютера до устройств и приложений. Разработчики провайдеров используют CIM для представления компонентов приложения, для которого они предусматривают возможность управления. Реализация CIM-представлений осуществляется на языке Managed Object Format (MOF).

Провайдер должен не только определять классы, представляющие объекты, но и обеспечивать WMI-интерфейс с объектами. WMI классифицирует провайдеры в соответствии с функциями их интерфейса. Эта классификация показана в таблице 4-10. Заметьте, что провайдер может реализовать несколько функций и благодаря этому выступать сразу в нескольких ролях, например провайдера классов и провайдера событий. Чтобы лучше понять определения функций в таблице 4-10, рассмотрим провайдер Event Log, реализующий несколько таких функций. Он поддерживает несколько объектов, включая Event Log Computer, Event Log Record и Event Log File. Event Log является провайдером Instance, так как способен определять несколько экземпляров своих классов. Один из классов, определяемых Event Log в нескольких экземплярах, — Event Log File (Win32_NTEventlogFile); экземпляр этого класса определяется для каждого журнала событий: System Event Log (Журнал системы), Application Event Log (Журнал приложений) и Security Event Log (Журнал безопасности).

Таблица 4-10. *Классификация провайдеров*

Классификация	Описание
Class (провайдер классов)	Предоставляет, изменяет, удаляет и перечисляет классы, специфичные для конкретного провайдера. Также может поддерживать обработку запросов. Примером этого редкого вида провайдеров служит Active Directory
Instance (провайдер экземпляров)	Предоставляет, изменяет, удаляет и перечисляет экземпляры системных или специфичных для провайдера классов. Каждый экземпляр представляет управляемый объект. Также может поддерживать обработку запросов
Property (провайдер свойств)	Предоставляет и изменяет значения свойств индивидуальных объектов
Method (провайдер методов)	Предоставляет методы класса, специфичного для провайдера
Event (провайдер событий)	Генерирует уведомления о событиях
Event consumer (провайдер потребителя событий)	Увязывает физического потребителя с логическим для поддержки уведомлений о событиях

Провайдер Event Log определяет данные экземпляра и позволяет управляющим приложениям перечислять записи. Методы backup и restore, реализуемые Event Log для объектов Event Log File, позволяют управляющим приложениям создавать резервные копии файлов Event Log и восстанавливать их через WMI. А это дает основания считать Event Log и провайдером Method. Наконец, управляющее приложение может зарегистрироваться на получение уведомлений о создании новых записей в файлах Event Log. Таким образом, Event Log, уведомляя WMI о создании новых записей, выступает еще и в роли провайдера Event.

CIM и язык Managed Object Format

CIM следует по стопам объектно-ориентированных языков типа C++ и Java, в которых средства моделирования создают представления в виде классов. Работая с классами, программисты могут использовать мощные методы моделирования, например наследование и включение. Подклассы могут наследовать атрибуты класса-предка, добавляя при этом собственные и даже переопределяя унаследованные. Класс, наследующий свойства другого класса, считается производным. В то же время классы можно составлять, создавая новый класс, включающий другие.

DMTF предоставляет набор классов как часть стандарта WBEM. Эти классы образуют базовый язык CIM и представляют объекты, применимые во всех сферах управления. Классы являются частью *базовой модели CIM* (CIM core model). Примером базового класса может служить CIM_ManagedSystemElement. У него есть несколько базовых свойств, идентифицирующих физические компоненты вроде аппаратных устройств и логические компоненты типа процессов и файлов. К свойствам относятся заголовок (caption),

описание (description), дата установки (installation date) и статус (status). Таким образом, классы CIM_LogicalElement и CIM_PhysicalElement наследуют атрибуты класса CIM_ManagedSystemElement. Эти два класса тоже входят в базовую модель CIM. В стандарте WBEM эти классы называются *абстрактными*, поскольку они существуют только как наследуемые другими классами (т. е. создать объект абстрактного класса нельзя). Абстрактные классы можно считать шаблонами, которые определяют свойства, используемые в других классах.

Вторая категория классов представляет объекты, специфичные для сфер управления, но независимые от конкретной реализации. Эти классы образуют *общую модель* (common model) и считаются расширением базовой модели. Пример класса общей модели — CIM_FileSystem, наследующий атрибуты CIM_LogicalElement. Поскольку практически все операционные системы, включая Windows, Linux и прочие вариации UNIX, опираются на хранилище данных, структурируемое на основе той или иной файловой системы, класс CIM_FileSystem является важной частью общей модели.

Последняя категория классов, *расширенная модель* (extended model), включает расширения, специфичные для конкретных технологий. В Windows определен обширный набор таких классов, представляющих объекты, специфичные для подсистемы Windows. Так как все операционные системы хранят данные в файлах, в общую модель CIM входит класс CIM_LogicalFile. Класс CIM_DataFile наследует свойства CIM_LogicalFile, а Windows добавляет классы Win32_PageFile и Win32_ShortcutFile для соответствующих типов файлов в подсистеме Windows.

Провайдер Event Log интенсивно использует наследование. На рис. 4-20 показано, как выглядит WMI CIM Studio, браузер классов, поставляемый с WMI Administrative Tools (этот набор можно бесплатно получить с сайта Microsoft). Использование наследования в провайдере Event Log можно наблюдать на примере его класса Win32_NTEventlogFile, производного от CIM_DataFile. Файлы Event Log — это файлы данных, которые имеют дополнительные атрибуты, специфичные для файлов журналов: имя файла журнала (LogfileName) и счетчик числа записей в файле (NumberOfRecords). Отображаемое браузером дерево классов показывает, что Win32_NTEventlogFile использует несколько уровней наследования: CIM_DataFile является производным от CIM_LogicalFile, тот — от CIM_LogicalElement, а последний — от CIM_ManagedSystemElement.

Как уже говорилось, разработчики провайдеров WMI пишут свои классы на языке MOF. Ниже показано определение класса Win32_NTEventlogFile компонента Event Log, выбранного на рис. 4-20. Обратите внимание на корреляцию свойств, перечисленных в правой секции окна браузера классов, и их определений. Свойства, наследуемые классом, помечаются в CIM Studio желтыми стрелками, и в определении Win32_NTEventlogFile эти свойства отсутствуют.

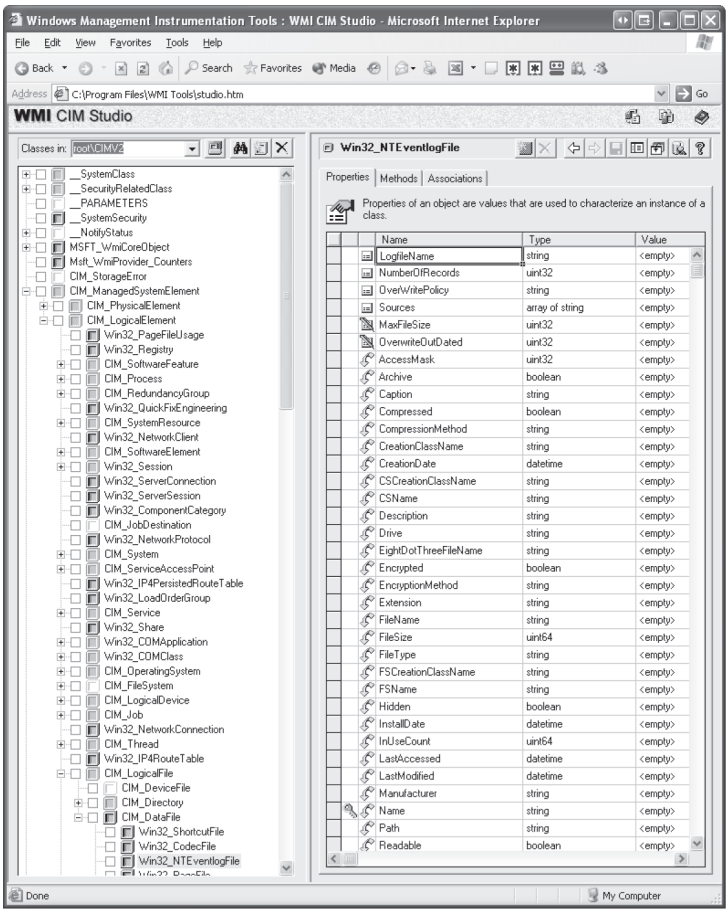


Рис. 4-20. *WMI CIM Studio — браузер классов*

```
[dynamic: ToInstance, provider("MS_NT_EVENTLOG_PROVIDER"),
Locale(1033), UUID("{8502C57B-5FBB-11D2-AAC1-006008C78BC7}")]
class Win32_NTEventlogFile : CIM_DataFile
{
   [read] string LogfileName;
   [read,write] uint32 MaxFileSize;
   [read] uint32 NumberOfRecords;
   [read,volatile,ValueMap{"0", "1..365", "4294967295"}]
      string OverWritePolicy;
   [read,write,Units("Days"),Range("0-365 | 4294967295")]
      uint32 OverwriteOutDated;
   [read] string Sources[];
   [implemented,Privileges{"SeSecurityPrivilege",
     "SeBackupPrivilege"}]
      uint32 ClearEventlog([in] string ArchiveFileName);
   [implemented,Privileges{"SeSecurityPrivilege",
```

```
      "SeBackupPrivilege"}]
      uint32 BackupEventlog([in] string ArchiveFileName);
};
```

Одно ключевое слово, на которое стоит обратить внимание в заголовке класса Win32_NTEventlogFile, — *dynamic*. Его смысл в том, что всякий раз, когда управляющее приложение запрашивает свойства объекта, инфраструктура WMI обращается к WMI-провайдеру за значениями соответствующих свойств, сопоставленных с объектом данного класса. *Статическим* (static) считается класс, находящийся в репозитарии WMI; в этом случае инфраструктура WMI получает значения свойств из репозитария и не обращается к WMI-провайдеру. Поскольку обновление репозитария — операция относительно медленная, динамические компоненты доступа более эффективны в случае объектов с часто изменяемыми свойствами.

ЭКСПЕРИМЕНТ: просмотр MOF-определений WMI-классов

Для просмотра MOF-определения любого WMI-класса используйте утилиту WbemTest, поставляемую с Windows. В этом эксперименте мы покажем, как увидеть MOF-определение класса Win32_NTEventLogFile.

1. Запустите Wbemtest через диалоговое окно Run (Запуск программы), открываемое из меню Start (Пуск).
2. Щелкните кнопку Connect (Подключить), измените Namespace (Namespace) на root\cimv2 и вновь щелкните кнопку Connect.
3. Выберите Enum Classes (Классы), установите переключатель Recursive (Рекурсивно) и нажмите OK.
4. Найдите Win32_NTEventLogFile в списке классов и дважды щелкните его, чтобы увидеть свойства этого класса.
5. Щелкните кнопку Show MOF (Вывести MOF), чтобы открыть окно с MOF-определением.

После создания классов на MOF разработчики могут предоставлять их определения в WMI несколькими способами. Разработчик провайдера компилирует MOF-файл в двоичный (BMF), более компактную форму представления, и передает BMF-файл инфраструктуре WMI. Другой способ заключается в компиляции MOF-файла и программной передаче определений от провайдера в инфраструктуру WMI через функции WMI COM API. Наконец, можно задействовать утилиту MOF Compiler (Mofcomp.exe), чтобы передать скомпилированное представление классов непосредственно инфраструктуре WMI.

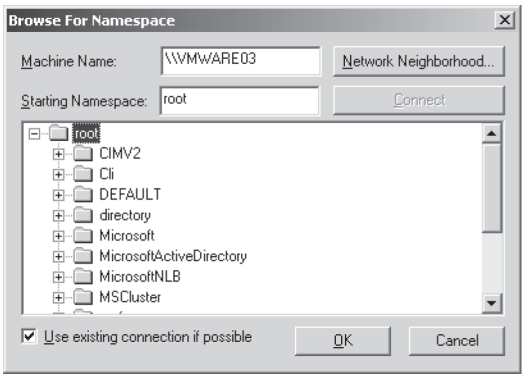
Пространство имен WMI

Классы определяют свойства объектов, а объекты являются экземплярами классов в системе. Для иерархического упорядочения объектов WMI использует пространство имен, в котором может содержаться несколько подпространств имен. Управляющее приложение должно подключиться к пространству имен, прежде чем оно сможет получить доступ к расположенным там объектам.

Корневой каталог пространства имен WMI называется *корнем* и обозначается как Root. В каждой WMI-системе есть четыре предопределенных пространства имен, расположенных под корнем: CIMV2, Default, Security и WMI. Некоторые из них тоже включают другие пространства. Так, в CIMV2 входят подпространства имен Applications и ms_409. Иногда провайдеры определяют собственные пространства имен, например в Windows можно увидеть пространство имен WMI (определяемое WMI-провайдером для драйверов устройств).

ЭКСПЕРИМЕНТ: просмотр пространств имен WMI

Увидеть, какие пространства имен определены в системе, позволяет WMI CIM Studio. Этот браузер открывает при запуске диалоговое окно подключения, в котором справа от поля ввода пространства имен имеется кнопка для просмотра пространств имен. Выбрав интересующее вас пространство имен, вы заставите WMI CIM Studio подключиться к этому пространству имен. В Windows Server 2003 под корнем определено свыше десятка пространств имен, некоторые из которых видны на следующей иллюстрации.



В отличие от пространства имен файловой системы, которое включает иерархию каталогов и файлов, пространство имен WMI имеет только один уровень вложения. Вместо имен WMI использует свойства объектов, которые определяет как *ключи* (keys), идентифицирующие эти объекты. Указывая объект в пространстве имен, управляющие приложения сообщают имя класса и ключ. Таким образом, каждый экземпляр класса уникально идентифицируется его ключом. Например, компонент доступа Event Log представляет записи в журнале событий классом Win32_NTLogEvent. У этого класса есть два ключа: Logfile (строковый) и RecordNumber (беззнаковый целочисленный). Поэтому, запрашивая у WMI экземпляры записей журнала событий, управляющее приложение идентифицирует их парой ключей. Вот пример ссылки на одну из записей:

```
\\DARYL\root\CIMV2:Win32_NTLogEvent.Logfile="Application",
                                    RecordNumber="1"
```

Первая часть имени (\\DARYL) идентифицирует компьютер, на котором находится объект, а вторая (\root\CIMV2) — пространство имен, где размещен объект. Имя класса следует после двоеточия, а имена ключей и их значения — после точки. Значения ключей отделяются запятыми.

WMI предоставляет интерфейсы, позволяющие приложениям перечислять все объекты конкретного класса или выполнять запросы, которые возвращают экземпляры какого-либо класса, удовлетворяющие критериям запроса.

Классы сопоставления

Многие типы объектов так или иначе связаны между собой. Например, объект «компьютер» включает объекты «процессор», «программное обеспечение», «операционная система», «активный процесс» и т. д. WMI позволяет создавать *класс сопоставления* (association class), представляющий логическую связь между двумя классами и поэтому имеющий всего два свойства: имя класса и модификатор *Ref*. Ниже показан исходный текст на MOF, сопоставляющий классы Win32_NTLogEvent и Win32_ComputerSystem. Получив какой-то объект, управляющее приложение может запрашивать и сопоставленные объекты. Благодаря таким сопоставлениям компонент доступа получает возможность определять иерархию объектов.

```
[dynamic: ToInstance, provider("MS_NT_EVENTLOG_PROVIDER"):
ToInstance, EnumPrivileges{"SeSecurityPrivilege"}: ToSubClass,
Locale(1033): ToInstance, UUID("{8502C57F-5FBB-11D2-AAC1-
006008C78BC7}"): ToInstance, Association : DisableOverride
ToInstance ToSubClass]
class Win32_NTLogEventComputer
{
    [key, read: ToSubClass] Win32_ComputerSystem ref Computer;
    [key, read: ToSubClass] Win32_NTLogEvent ref Record;
};
```

На рис. 4-21 показан WMI Object Browser (еще один инструмент, включаемый в WMI Administrative Tools), который показывает содержимое пространства имен CIMV2. В это пространство имен обычно помещают свои объекты системные компоненты Windows. Браузер объектов сначала определяет местонахождение объекта MR-XEON, экземпляра Win32_ComputerSystem, представляющего компьютер. Далее браузер получает объекты, сопоставленные с Win32_ComputerSystem и отображает их под MR-XEON. Пользовательский интерфейс браузера объектов помечает сопоставленные объекты значком папки с двуглавой стрелкой.

Как видите, класс сопоставления Win32_NTLogEventComputer показывается под MR-XEON и существует несколько экземпляров класса Win32_NTLogEvent. Посмотрите на предыдущий листинг — вы убедитесь, что класс Win32_NTLogEventComputer определен для сопоставления классов Win32_ ComputerSystem и Win32_NTLogEvent. Выбрав в браузере объектов экземпляр Win32_NTLogEvent, вы увидите в правой секции на вкладке Properties свойства этого класса. Microsoft предоставляет WMI Object Browser, чтобы WMI-разработчики

могли изучать свои объекты, но управляющие приложения, выполняя те же операции, показывают свойства или собранные данные более наглядно.

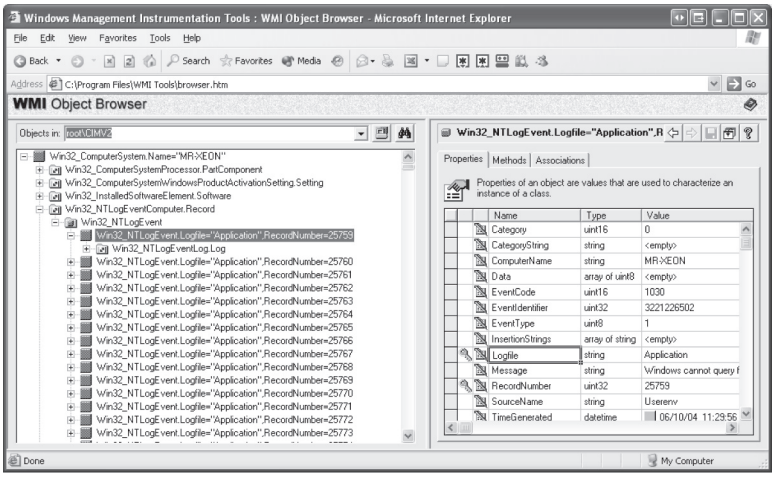


Рис. 4-21. *WMI Object Browser*

ЭКСПЕРИМЕНТ: использование WMI-сценариев для управления системами

Сильная сторона WMI — его поддержка языков сценариев. Microsoft создала сотни сценариев, выполняющих распространенные административные задачи для управления учетными записями пользователей, файлами, реестром, процессами и аппаратными устройствами. Некоторые сценарии поставляются с ресурсами Windows, но основная их часть находится на сайте Microsoft TechNet Scripting Center. Использовать сценарий с этого сайта очень легко: достаточно скопировать его текст из окна Интернет-браузера, сохранить в файле с расширением .vbs и запустить командой **cscript *script*.vbs**, где *script* — имя, присвоенное вами данному сценарию. Cscript — это интерфейс командной строки для Windows Script Host (WSH).

Вот пример сценария из TechNet, который регистрируется на получение событий при создании экземпляров Win32_Process (его экземпляр создается всякий раз, когда запускается какой-либо процесс) и выводит строку с именем процесса, представляемым данным объектом:

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:" _
    & "{impersonationLevel=impersonate}!\\" & _
    strComputer & "\root\cimv2")
Set colMonitoredProcesses = objWMIService. _
    ExecNotificationQuery("select * from _
    __instancecreationevent " _
    & " within 1 where TargetInstance isa 'Win32_Process'")
i = 0
```

см. след. стр.

```
Do While i = 0
    Set objLatestProcess = colMonitoredProcesses.NextEvent
    Wscript.Echo objLatestProcess.TargetInstance.Name
Loop
```

В строке, где вызывается *ExecNotificationQuery*, этой функции передается параметр, который включает выражение select из поддерживаемого WMI подмножества ANSI-стандарта Structured Query Language (SQL) только для чтения. Это подмножество называется WQL, и оно предоставляет WMI-потребителям гибкий способ задания информации, которую им нужно запросить от WMI-провайдеров. Если вы запустите этот сценарий с помощью Cscript, а затем запустите Notepad, то получите следующий вывод:

```
C:\>cscript monproc.vbs
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001.
All rights reserved.

NOTEPAD.EXE
```

Реализация WMI

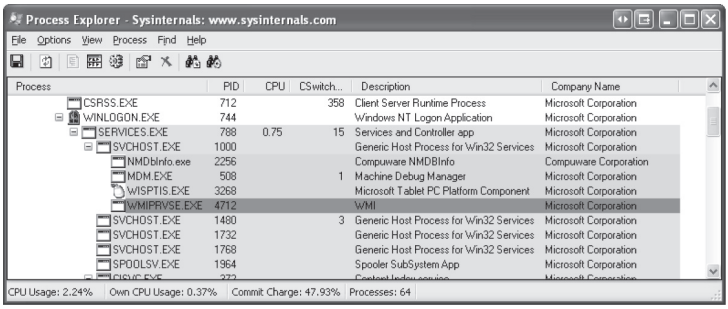
В Windows 2000 WMI-сервис реализован в \Windows\System32\Winmgmt.exe, который запускается SCM при первой попытке доступа управляющего приложения или WMI-провайдера к WMI API. В Windows XP и Windows Server 2003 WMI-сервис работает в общем процессе Svchost, выполняемом под учетной записью локальной системы.

В Windows 2000 WMI загружает провайдеры как внутренние (внутрипроцессные) DCOM-серверы, выполняемые внутри сервисного процесса Winmgmt. Если ошибка в провайдере приведет к краху процесса WMI, WMI-сервис завершится, а затем перезапустится в ответ на следующий запрос к WMI. Поскольку WMI-сервис разделяет свой процесс Svchost с несколькими другими сервисами, которые тоже могут завершаться при ошибке в WMI-провайдере, вызывающей закрытие этого процесса, в Windows XP и Windows Server 2003 WMI загружает провайдеры в хост-процесс Wmiprvse.exe. Он запускается как дочерний по отношению к сервисному процессу RPC. WMI выполняет Wmiprvse под учетной записью Local System, Local Service или Network Service в зависимости от значения свойства HostingModel экземпляра WMI-объекта Win32Provider, который представляет реализацию провайдера. Процесс Wmiprvse завершается, как только провайдер удаляется из кэша, спустя минуту после приема последнего запроса к провайдеру.

ЭКСПЕРИМЕНТ: наблюдение за созданием Wmiprvse

Чтобы понаблюдать за созданием Wmiprvse, запустите Process Explorer и выполните Wmic. Процесс Wmiprvse появится под процессом Svc-

host, который служит хостом сервиса RPC. Если в Process Explorer включена функция выделения заданий, вы увидите, что появился не только новый процесс, но и новое задание. Дело здесь вот в чем. Чтобы предотвратить исчерпание всей виртуальной памяти в системе плохо написанным провайдером, Wmiprvse запускается в объекте «задание», который ограничивает количество создаваемых дочерних процессов и объемы виртуальной памяти, допустимые для выделения каждым процессом. (Об объектах «задание» см. главу 6.)



Большинство компонентов WMI, в том числе MOF-файлы, DLL встроенных провайдеров и DLL управляющих приложений, по умолчанию размещаются в каталогах \Windows\System32 и \Windows\System32\Wbem. Во втором каталоге вы найдете MOF-файл провайдера Event Log, Ntevt.mof. Там же находится и Ntevt.dll, DLL провайдера Event Log, используемая WMI-сервисом.

В подкаталогах каталога \Windows\System32\Wbem находятся репозитарий, файлы журналов и MOF-файлы сторонних разработчиков. Репозитарий, называемый репозитарием CIMOM, реализуется в WMI с применением закрытой версии ядра баз данных Microsoft JET. В Windows 2000 база данных хранится в файле \Windows\System32\Wbem\Repository\Cim.rep. WMI учитывает параметры реестра (включая различные внутренние параметры в Windows 2000 вроде расположения резервных копий файлов CIMOM и интервалов между их созданием), которые хранятся в разделе HKLM\SOFTWARE\Microsoft\WBEM\CIMOM.

Для обмена данными с WMI и приема команд от него драйверы устройств используют специальные интерфейсы под общим названием WMI System Control Commands. Эти межплатформенные интерфейсы являются частью WDM (см. главу 9).

WMIC

В состав Windows XP и Windows Server 2003 входит утилита Wmic.exe, позволяющая взаимодействовать с WMI из оболочки командной строки с поддержкой WMI. Через эту оболочку доступны все WMI-объекты и их свойства и методы, что превращает WMIC в консоль расширенного управления системами.

Защита WMI

WMI реализует защиту на уровне пространства имен. Если управляющее приложение успешно подключилось к пространству имен, оно получает доступ к любым свойствам всех объектов этого пространства имен. Для управления доступом пользователей к пространству имен администратор может задействовать приложение WMI Control. Для запуска WMI Control последовательно откройте в меню Start (Пуск) подменю Programs (Программы) и Administrative Tools (Администрирование), а затем выберите команду Computer Management (Управление компьютером). Далее раскройте узел Services And Applications (Службы и приложения), щелкните правой кнопкой мыши строку WMI Control (Управляющий элемент WMI) и выберите команду Properties (Свойства) для вывода диалогового окна WMI Control Properties (Свойства: Управляющий элемент WMI), которое показано на рис. 4-22. Для настройки параметров защиты пространства имен перейдите на вкладку Security (Безопасность), выберите пространство имен и щелкните кнопку Security (Безопасность). Другие вкладки диалогового окна WMI Control Properties позволяют изменять сохраняемые в реестре настройки, которые относятся к функционированию WMI и резервному копированию.

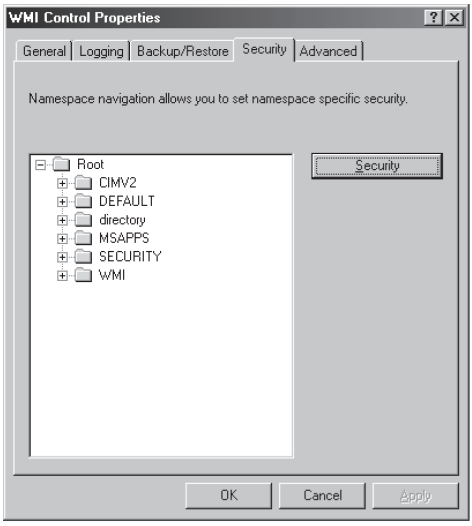


Рис. 4-22. *Параметры защиты WMI*

Резюме

К этому моменту мы уже рассмотрели общую структуру Windows, базовые системные механизмы, на которые опирается эта структура, и основные механизмы управления. Заложив такой фундамент, можно переходить к более подробному изучению процесса загрузки и индивидуальных компонентов исполнительной системы.

ГЛАВА 5

Запуск и завершение работы системы

В этой главе описываются стадии загрузки Windows, а также параметры, влияющие на процесс ее запуска. Понимание тонкостей процесса загрузки поможет вам в диагностике проблем, возникающих при загрузке Windows. Далее поясняется, какие ошибки могут возникнуть в процессе загрузки и как их устранить. В заключение мы рассмотрим, что происходит при корректном завершении работы системы.

Процесс загрузки

Описание процесса загрузки мы начнем с рассмотрения установки Windows, а затем исследуем выполнение загрузочных файлов. Поскольку драйверы устройств играют ключевую роль в процессе загрузки, будет уделено внимание и тому, как они контролируют собственную загрузку и инициализацию. Далее мы поясним, как инициализируются компоненты исполнительной системы и как ядро запускает пользовательскую часть Windows, активизируя процессы Session Manager (Smss.exe) и Winlogon, а также подсистему Windows. Попутно вы узнаете, что происходит внутри системы на момент вывода тех или иных текстовых сообщений, появляющихся на экране в процессе загрузки.

Ранние стадии процесса загрузки на платформах x86 и x64 сильно отличаются от таковых на платформе IA64. В следующих разделах описываются стадии, специфичные для x86 и x64, а потом поясняются стадии, специфичные для IA64.

Что предшествует загрузке на платформах x86 и x64

Процесс загрузки Windows начинается не при включении компьютера или нажатии кнопки Reset, а еще при установке этой системы на компьютер. На одном из этапов работы программы установки Windows (Windows Setup) происходит подготовка основного жесткого диска системы: на нем размещается код, в дальнейшем участвующий в процессах загрузки. Прежде чем рассказывать, что делает этот код, мы покажем, как и в какой области жесткого диска он размещается.

Стандарт разбиения физических жестких дисков на тома существует в системах типа x86 со времен первых версий MS-DOS. Операционные системы Microsoft разбивают жесткие диски на дискретные области, называемые *разделами* (partitions). После форматирования с использованием файловых систем (типа FAT и NTFS) разделы образуют тома. На жестком диске может быть до четырех главных разделов (primary partitions). Поскольку это ограничило бы количество томов на одном диске, данная схема предусматривает особый тип раздела — *дополнительный* (extended partition), что дает до четырех дополнительных разделов в главном разделе. Дополнительные разделы могут содержать другие дополнительные разделы, те в свою очередь — третьи дополнительные разделы и т. д. Так что диск можно разбить практически на бесконечное число томов. Пример разбиения жесткого диска на разделы показан на рис. 5-1, а компоненты, участвующие в процессе загрузки, перечислены в таблице 5-1. (Подробнее о разбиении жестких дисков в Windows см. главу 10.)

Таблица 5-1. *Компоненты, участвующие в процессе загрузки на платформах x86 и x64*

Компонент	Режим работы	Описание
Код главной загрузочной записи (master boot record, MBR)	16-разрядный реальный	Считывает в память загрузочные секторы раздела
Загрузочный сектор	16-разрядный реальный	Считывает корневой каталог для загрузки Ntldr
Ntldr	16-разрядный реальный и 32- или 64-разрядный защищенный (активизирует поддержку подкачки страниц)	Считывает Boot.ini, выводит загрузочное меню, загружает Ntoskrnl.exe, Bootvid.dll, Hal.dll и драйверы устройств, необходимые для загрузки и последующего запуска системы. Если загружается 32-разрядная система, переключает в 32-разрядный защищенный режим, а если загружается 64-разрядная система — в 64-разрядный защищенный режим
Ntdetect.com	16-разрядный реальный	Обеспечивает распознавание устройств для Ntldr
Ntbootdd.sys	Защищенный	Драйвер устройства, применяемый для дисковых операций ввода-вывода в системах со SCSI и ATA (Advanced Technology Attachment), где BIOS не используется
Ntoskrnl.exe	Защищенный с поддержкой подкачки страниц	Инициализирует компоненты исполнительной системы, драйверы, необходимые для загрузки и запуска системы, подготавливает систему к выполнению встроенных приложений и запускает Smss.exe

Таблица 5-1. (*окончание*)

Компонент	Режим работы	
Smss	Встроенное приложение	Загружает подсистему Windows, включая Win32k.sys и Csrss.exe, и запускает процесс Winlogon
Winlogon	Встроенное приложение	Загружает SCM, подсистему локальной аутентификации (LSASS) и выводит на экран диалоговое окно для входа в систему
Диспетчер управления сервисами (service control manager, SCM)	Встроенное приложение	Загружает и инициализирует автоматически запускаемые драйверы устройств и сервисы Windows

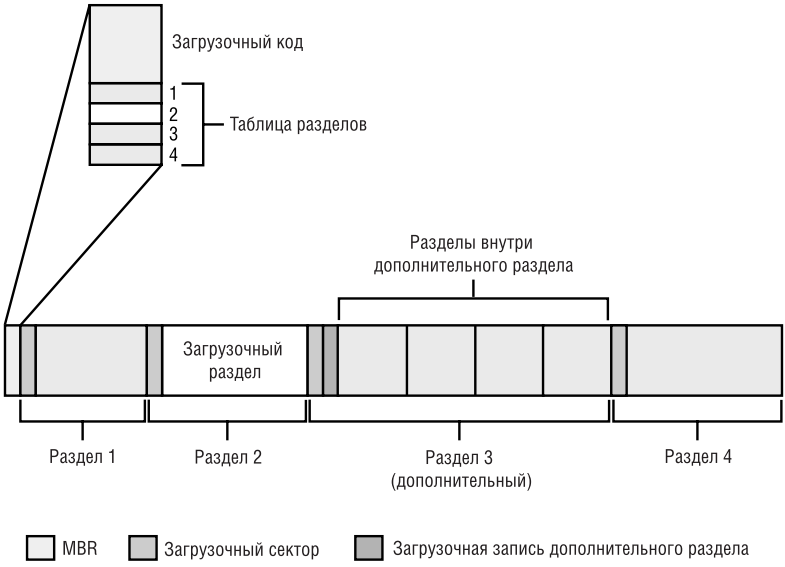


Рис. 5-1. *Пример структуры разделов жесткого диска*

Единицей адресации физических дисков является *сектор*. Типичный размер сектора жесткого диска на IBM-совместимом PC — 512 байтов. Такие утилиты, как Fdisk в MS-DOS или программа Windows Setup, позволяющие создавать на жестком диске тома, записывают в первый сектор жесткого диска специальные данные, создавая таким образом главную загрузочную запись (MBR) диска (детали см. в главе 10). Размер MBR фиксирован. Она состоит из набора машинных команд (загрузочный код) и таблицы разделов с четырьмя записями, которые определяют расположение главных разделов на диске. Первый код, выполняемый при загрузке IBM-совместимого компьютера, называется BIOS, — он хранится в ПЗУ компьютера. BIOS вы-

бирает загрузочное устройство, считывает его MBR в память и передает управление ее загрузочному коду.

MBR начинает со сканирования таблицы разделов в поисках раздела, помеченного особым флагом. Этот флаг сигнализирует, что данный раздел является загрузочным. Как только MBR обнаружит хотя бы один такой флаг, она считывает в память код из первого сектора раздела, помеченного флагом, и передает ему управление. Такой раздел называется *загрузочным*, как и его первый сектор, а том, определенный для загрузочного раздела, — *системным*.

Операционная система, как правило, ведет запись в загрузочные сектора без участия пользователя. Например, Windows Setup при записи MBR одновременно создает в первом загрузочном разделе жесткого диска свой загрузочный сектор. Перед этим программа установки проверяет, является ли он сейчас загрузочным сектором MS-DOS. Если да, Windows Setup сначала копирует содержимое загрузочного сектора в файл Bootsect.dos, помещая его в корневой каталог раздела.

Перед записью в загрузочный сектор Windows Setup проверяет совместимость текущей файловой системы этого раздела с Windows. В любом случае она может отформатировать данный раздел с использованием выбранной вами файловой системы (FAT, FAT32 или NTFS). Если раздел уже отформатирован, вы можете пропустить этот этап. После того как загрузочный раздел отформатирован, Setup копирует на него файлы Windows, в том числе два стартовых файла, Ntldr и Ntdetect.com.

Еще одна задача программы установки — создание файла загрузочного меню, Boot.ini, в корневом каталоге системного тома. В этом файле содержатся параметры запуска устанавливаемой версии Windows, а также сведения обо всех системах, установленных до Windows. Если файл Bootsect.dos содержит загрузочный сектор MS-DOS, в Boot.ini добавляется запись, позволяющая загружать MS-DOS. Ниже приведен пример файла Boot.ini с поддержкой двухвариантной загрузки для компьютера, на котором перед установкой Windows XP была установлена MS-DOS.

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)
\WINDOWS="Microsoft Windows XP Professional " /fastdetect
C:\="Microsoft Windows"
```

Заметьте, что в этом примере путь к каталогу Windows задан по специальному синтаксису, отвечающему соглашению по именованию Advanced RISC Computing (ARC). В Windows используется три вида такого синтаксиса. Первый, синтаксис multi(), показан в примере; он указывает Windows загружать системные файлы через функции прерывания INT 13, предоставляемые BIOS. Таким образом, синтаксис multi() применяется, когда у диска, на котором находится загрузочный том, есть контроллер с поддержкой прерывания INT 13. Синтаксис multi() имеет следующий формат:

```
multi(W)disk(X)rdisk(Y)partition(Z)
```

где *W* — номер дискового контроллера (также называемый порядковым номером), обычно равный 0, *X* — всегда 0 в синтаксисе multi(), а *Y* указывает физический жесткий диск, подключенный к контроллеру *W*. Для ATA-контроллеров значение *Y*, как правило, укладывается в диапазон от 0 до 3, для SCSI-контроллеров — 0–15. *Z* сообщает номер раздела на физическом диске, соответствующего загрузочному тому. Первому разделу присваивается значение 1.

ARC-синтаксис scsi() сообщает Windows, что для доступа к файлам на загрузочном томе нужно задействовать сервисы дискового ввода-вывода, предоставляемые Ntbootdd.sys (о нем чуть позже). Синтаксис scsi() выглядит так:

```
scsi(W)disk(X)rdisk(Y)partition(Z)
```

Здесь *W* — номер контроллера, *X* — физический жесткий диск, подключенный к этому контроллеру (обычно равен 0–15). *Y* указывает SCSI LUN-номер (logical unit number) диска, содержащего загрузочный том, и, как правило, равен 0. Наконец, *Z* — это раздел, соответствующий загрузочному тому с нумерацией, начинающейся от 1.

Наконец, в Windows применяется третий вид синтаксиса — signature(). Он указывает Windows найти диск с сигнатурой, соответствующей первому значению в скобках, независимо от номера контроллера, сопоставленного с этим диском, и использовать Ntbootdd.sys для доступа к загрузочному тому. *Сигнатура диска* (disk signature) — это глобально уникальный идентификатор (GUID), извлекаемый Windows Setup из информации в MBR и записываемый на диск. Синтаксис signature() выглядит так:

```
signature(V)disk(X)rdisk(Y)partition(Z)
```

где *V* — 32-битная сигнатура диска в шестнадцатеричной форме, идентифицирующая диск, *X* — физический жесткий диск со специфической сигнатурой, который может быть подключен к любому контроллеру в системе, *Y* — всегда 0, а *Z* — номер раздела, на котором находится загрузочный том.

Windows использует синтаксис signature(), если:

- размер загрузочного тома больше 7,8 Гб, а BIOS-функции расширенного прерывания INT 13 (которые применяются для доступа к частям диска за пределами 7,8 Гб) не могут обращаться ко всему тому;
- BIOS не поддерживает расширенное прерывание INT 13.

Загрузочный сектор и Ntldr на платформах x86 и x64

Перед тем как произвести запись в загрузочный сектор, программа установки должна выяснить формат раздела, поскольку от него зависит содержимое загрузочного сектора. Если это раздел FAT, Windows записывает в загрузочный сектор код, поддерживающий файловую систему FAT. Если раздел отформатирован под NTFS, в загрузочный сектор записывается код, соответ-

ствующий NTFS. Задача кода загрузочного сектора — предоставлять Windows информацию о структуре и формате тома и считывать из его корневого каталога файл Ntldr. После считывания Ntldr в память код загрузочного сектора передает управление в точку входа Ntldr. Если код загрузочного сектора не может найти Ntldr в корневом каталоге тома, он выводит сообщение об ошибке: «BOOT: Couldn't find NTLDRP» (в FAT) или «NTLDR is missing» (в NTFS).

Ntldr начинает свою работу, когда система функционирует в реальном режиме (real mode) x86. В реальном режиме трансляция между виртуальными и физическими адресами не осуществляется, поэтому программы, использующие какие-либо адреса памяти, интерпретируют их как физические. В этом режиме доступен лишь первый мегабайт физической памяти компьютера; в нем выполняются простые программы MS-DOS. Однако первое, что делает Ntldr, — переключает систему в защищенный режим (protected mode). На этой стадии трансляция между виртуальными адресами и физическими по-прежнему отсутствует, но становится доступным полный объем памяти. Переключив систему в защищенный режим, Ntldr может работать со всей физической памятью. После того как он создает таблицы страниц, число которых достаточно для доступа к нижним 16 Мб памяти с подкачкой, Ntldr включает поддержку подкачки страниц. Защищенный режим с подкачкой страниц является нормальным режимом работы Windows.

С этого момента Ntldr может работать в полнофункциональном режиме. Но при доступе к IDE-дискам и дисплею Ntldr все еще зависит от функций загрузочного кода, которые на непродолжительное время отключают подкачку страниц и возвращают процессор в режим, позволяющий выполнять сервисы BIOS. Если диск, содержащий загрузочный или системный том, является SCSI-устройством и недоступен через BIOS, Ntldr загружает файл Ntbootdd.sys и использует его функции доступа к диску вместо аналогичных функций загрузочного кода. Ntbootdd.sys — это экземпляр минипорт-драйвера SCSI, применяемый Windows для полноценного доступа к загрузочному диску. (О дисковых драйверах см. главу 10.) Затем Ntldr с помощью встроенного кода файловой системы считывает из корневого каталога файл Boot.ini. В отличие от кода загрузочного сектора код Ntldr способен читать и подкаталоги.

Далее Ntldr очищает экран. Если в корневом каталоге системного тома присутствует допустимый файл Hiberfil.sys, Ntldr считывает его содержимое в память и передает управление коду в ядре, восстанавливающему спящую (hibernated) систему. Этот код отвечает за перезапуск драйверов, которые были активны на момент выключения системы. Hiberfil.sys считается допустимым, только если при последнем выключении компьютер был переведен в спящий режим. (О спящем режиме см. раздел «Диспетчер электропитания» главы 11.)

Если в файле Boot.ini имеется более одной записи о доступных для загрузки операционных системах, Ntldr выводит загрузочное меню. (Если в файле Boot.ini только одна запись, Ntldr пропускает загрузочное меню и сразу выводит стартовый индикатор прогресса загрузки.) Информация из Boot.ini адресует Ntldr к разделу, в котором находится системный каталог Windows

(обычно \Windows). Этим разделом может быть как загрузочный, так и другой главный раздел.

Если запись Boot.ini ссылается на MS-DOS, Ntldr считывает в память содержимое файла Bootsect.dos, переключается обратно в 16-разрядный реальный режим и вызывает из Bootsect.dos код MBR. В результате код из Bootsect.dos выполняется аналогично коду, считанному MBR с диска. Код из Bootsect.dos инициирует процесс загрузки, специфичный для MS-DOS. Так же происходит загрузка Microsoft Windows Me, Windows 98 или Windows 95, если они установлены вместе с Windows.

Записи Boot.ini могут включать ряд необязательных параметров, интерпретируемых Ntldr и другими компонентами в процессе загрузки. Полный список этих параметров приводится в таблице 5-2. Утилита Bootcfg.exe, впервые появившаяся в Windows XP, предоставляет удобный интерфейс для задания ряда параметров. Любые параметры, включаемые в Boot.ini, сохраняются в параметре реестра HKLM\System\CurrentControlSet\Control\SystemStartOptions.

Таблица 5-2. *Параметры в Boot.ini*

Параметр	Описание
/3GB только для 32-разрядных систем)	Увеличивает пользовательскую часть адресного пространства процессов с 2 до 3 Гб, тем самым уменьшая размер системной части пространства с 2 до 1 Гб. Позволяет повысить производительность приложений, интенсивно использующих виртуальную память (например, серверов баз данных), предоставляя им большее адресное пространство. Но здесь нужно соблюсти еще два условия: система должна работать под управлением Windows XP, Windows Server 2003, Windows 2000 Advanced Server или Datacenter Server, а в исполняемом файле приложения должен присутствовать флаг, указывающий на поддержку 3-гигабайтного пользовательского адресного пространства (см. раздел «Структура адресного пространства» главы 7)
/BASEVIDEO	Заставляет Windows использовать стандартный драйвер VGA-видеоадаптера для работы в GUI-режиме
/BAUDRATE=	Включает отладку в режиме ядра и позволяет изменить устанавливаемую по умолчанию скорость передачи по соединению с удаленным хостом, используемым для отладки ядра (19200). Например: /BAUDRATE=115200
/BOOTLOG	Заставляет Windows вести журнал загрузки и записывать его в файл %SystemRoot%\Ntbtlog.txt
/BOOTLOGO	Позволяет отображать собственный экран-заставку при загрузке Windows XP или Windows Server 2003 вместо стандартного. Для этого сначала создайте 16-цветное (любые 16 цветов) растровое изображение размером 640x480 и сохраните его в каталоге Windows под именем Boot.bmp. Затем добавьте /bootlogo/noguiboot в boot.ini (там, где осуществляется выбор)

см. след. стр.

Таблица 5-2. (*продолжение*)

Параметр	Описание
/BREAK	Вызывает остановку HAL на точке прерывания в процессе инициализации. Первое, что делает ядро Windows, — инициализирует HAL. Так что данная точка прерывания является первой из возможных. HAL может неопределенно долго ждать соединения с удаленным отладчиком ядра. При использовании этого параметра без /DEBUG появляется «синий экран» со стоп-кодом 0x00000078 (PHASE0_EXCEPTION)
/BURNMEMORY=	Указывает объем памяти, запрещаемой для использования операционной системой Windows (по аналогии с параметром /MAXMEM). Значение задается в мегабайтах. Например, /BURNMEMORY=128 означает, что 128 Мб физической памяти компьютера недоступны Windows
/CHANNEL=	Используется в сочетании с /DEBUGPORT=1394, чтобы задать канал IEEE 1394, через который осуществляется коммуникационное взаимодействие при удаленной отладке ядра. Принимает любое значение из диапазона 0–62 и по умолчанию (если не указано конкретное значение) равен 0
/CLKLVL	Перенастраивает стандартную многопроцессорную версию HAL для систем типа x86 (Halmps.dll) на распознавание сигналов системного таймера по потенциалу, а не по фронту
/CMDCONS	Передается при загрузке консоли восстановления (Recovery Console)
/CRASHDEBUG	При загрузке системы загружает и отладчик ядра, который остается неактивным до момента краха. Этот параметр освобождает последовательный порт, который иначе был бы постоянно задействован отладчиком. Пока не произошел крах, порт может использоваться системой (параметр /DEBUG, напротив, заставляет отладчик ядра постоянно занимать последовательный порт)
/DEBUG	Включает отладку в режиме ядра
/DEBUGPORT=	Включает отладку в режиме ядра и назначает последовательный порт для подключения удаленного хоста с отладчиком ядра, отличный от заданного по умолчанию (обычно COM2 в системах минимум с двумя последовательными портами). Windows XP и Windows Server 2003 поддерживают удаленную отладку и через порты IEEE 1394. Например: /DEBUGPORT=COM2, /DEBUGPORT=1394
/EXECUTE	Отключает защиту «запрет исполнения», см. также /NOEXECUTE
/FASTDETECT	Параметр загрузки Windows по умолчанию. Заменяет /NOSERIALMICE, применяемый в Windows NT 4. Введен для поддержки модулем NTDETECT альтернативной загрузки Windows NT 4. В Windows устройства, подключенные к параллельным и последовательным портам, определяются PnP-драйверами устройств, тогда как в Windows NT 4 эти функции возлагаются на NTDETECT. Параметр /FASTDETECT заставляет NTDETECT пропускать перечисление устройств, подключенных к параллельным и последовательным портам (не требуемое при загрузке Windows). А в отсутствие этого параметра NTDETECT перечисляет такие устройства, что необходимо для загрузки NT 4

Таблица 5-2. (*продолжение*)

Параметр	Описание
/INTAFFINITY	Указывает стандартной многопроцессорной версии HAL для систем типа x86 (Halmps.dll) так настроить привязку прерываний, чтобы лишь один процессор (с наибольшим порядковым номером) принимал запросы на прерывания. В отсутствие этого параметра (по умолчанию) HAL разрешает принимать запросы на прерывания всем процессорам
/KERNEL= /HAL=	Позволяет задавать имена файлов образа ядра и/или HAL, отличные от используемых по умолчанию (Ntoskrnl.exe и Hal.dll). Эти параметры предназначены для переключения между проверочными (отладочными) и рабочими версиями ядра, а также для выбора HAL вручную. Если вы хотите загрузить отладочную среду, состоящую только из проверочных версий ядра и HAL (этого, как правило, достаточно для тестирования драйверов), выполните следующие операции. 1. Скопируйте отладочные версии ядра с дистрибутивного компакт-диска в каталог \Windows\System32, изменив при этом их имена. Например, на однопроцессорной системе скопируйте Ntoskrnl.exe в Ntoschk.exe и Ntkrnlpa.exe в Ntoschkpa.exe, а на многопроцессорной — Ntkrnlmp.exe в Ntoschk.exe и Ntkrpamp.exe в Ntoschkpa.exe. Файлу ядра нужно присвоить краткое имя в формате «8.3». 2. Скопируйте с дистрибутивного компакт-диска \I386\Driver.cab подходящую проверочную версию HAL в файл Halchk.dll в каталоге \Windows\System32. Чтобы узнать, какую именно версию HAL нужно скопировать, найдите в \Windows\Repair\Setup.log строку с Hal.dll. Она выглядит примерно так: \Windows\system32\hal.dll=«halacpi.dll», «1d8a1». Искомое имя файла HAL находится сразу после знака равенства. Файлу HAL нужно присвоить краткое имя в формате «8.3». 3. Сделайте копию строк, присутствовавших в файле Boot.ini по умолчанию. 4. В строку с определением элементов загрузочного меню добавьте новый элемент для отладочной среды (например «Windows XP Professional Checked»). 5. В конец нового элемента добавьте /KERNEL=NTOSCHK.EXE /HAL= HALCHK.DLL. Учтите, что теперь в загрузочном меню доступны два элемента: новый — для загрузки отладочной среды и старый — для загрузки рабочей среды
/LASTKNOWN GOOD	Заставляет систему загружаться так, будто в загрузочном меню была выбрана команда загрузки последней удачной конфигурации
/MAXMEM=	Ограничивает объем памяти, доступный Windows. Физическая память, лежащая за пределами указанного значения, системой игнорируется (не используется). Значение задается в мегабайтах. Например, при /MAXMEM=32 система использует только первые 32 Мб физической памяти

см. след. стр.

Таблица 5-2. (*продолжение*)

Параметр	Описание
/MAXPROCSPER-CLUSTER=	При использовании стандартной многопроцессорной версии HAL для систем x86 (Halmps.dll) включает кластерный режим адресации контроллера прерываний APIC. Не поддерживается при использовании внешнего APIC-контроллера 82489DX
/MININT	Этот параметр используется Windows PE (Preinstallation Environment) и заставляет диспетчер конфигурации загрузить куст реестра SYSTEM как изменяемый (volatile), чтобы его изменения в памяти не сохранялись в соответствующем образе на диске
/NODEBUG	Запрещает инициализацию отладки режима ядра. Имеет больший приоритет, чем остальные параметры, применяемые для отладки (/DEBUG, /DEBUGPORT и /BAUDRATE)
/NOEXECUTE	Доступен только в 32-разрядных версиях Windows при выполнении на процессорах AMD64 и только при включенной поддержке PAE (см. также /PAE). Включает защиту «запрет исполнения» (no-execute protection), что заставляет диспетчер памяти помечать страницы с данными как не исполняемые, чтобы их нельзя было выполнять как код. Это полезно для предотвращения попыток зловредного кода использовать ошибки с переполнением буферов для запуска произвольного кода. Такая защита всегда включена в 64-разрядных версиях Windows на процессорах AMD64. Существует 4 модификатора, которые можно указать в /NOEXECUTE: =OPTIN, =OPTOUT, =ALWAYSON, =ALWAYSOFF. Подробности см. в главе 7
/NOGUIBOOT	Запрещает Windows инициализацию VGA-драйвера, ответственного за вывод растровой графики в процессе загрузки. Этот драйвер используется для вывода информации о ходе загрузки, поэтому при его отключении Windows не будет выводить эту информацию
/NOLOWMEM	Применяется только с параметром /PAE и только при наличии в системе более 4 Гб физической памяти. Если эти условия соблюдены, PAE-версия ядра Windows, Ntkrnlpa.exe, не использует первые 4 Гб физической памяти. Для загрузки драйверов и приложений, а также для создания всех пулов памяти используется область выше этой границы. Данный параметр предназначен для тестирования драйверов на совместимость с системами, в которых установлено более 4 Гб памяти
/NOPAE	Заставляет Ntldr загружать версию ядра, не поддерживающую PAE (Physical Address Extensions) даже в том случае, когда x86-система поддерживает такой механизм и имеет более 4 Гб физической памяти
/NOSERIALMICE=[COM*x*|COM*x*,*y*,*z*...]	Устаревший спецификатор Windows NT 4, замененный на /FASTDETECT. Отключает распознавание мыши на указанных COM-портах. Применяется в том случае, если к последовательному порту подключено устройство, отличное от мыши. При использовании /NOSERIALMICE без указания порта распознавание мыши отключается для всех COM-портов. См. также статью Q131976 в Microsoft Knowledge Base

Таблица 5-2. (*продолжение*)

Параметр	Описание
/NUMPROC=	Указывает число процессоров, которые Windows может использовать в многопроцессорной системе. Например, указав /NUMPROC=2 в четырехпроцессорной системе, вы запретите использование двух из четырех процессоров
/ONECPU	В многопроцессорных системах заставляет Windows работать только с одним процессором
/PAE	Указывает Ntldr загрузить Ntkrnlpa.exe, версию ядра (для x86-процессоров), использующую преимущества PAE. PAE обеспечивает 64-разрядную адресацию драйверами устройств. Этот параметр предназначен для тестирования драйверов на совместимость с системами с большим объемом памяти
/PCILOCK	Запрещает Windows динамически назначать PCI-устройствам IRQ и другие ресурсы для ввода-вывода. При этом используются настройки, заданные в BIOS. Подробности см. в статье Q148501 в Microsoft Knowledge Base
/RDPATH=	Указывает путь к файлу System Disk Image (SDI), который может находиться в сети и который система должна использовать для загрузки. Часто применяется в сочетании с флагом /RDIMAGEOFFSET=, сообщающим NTLDR, где в этом файле начинается образ системы
/REDIRECT	Введен в Windows Server 2003. Заставляет Windows включить поддержку сервисов аварийного управления (Emergency Management Services, EMS), которые выводят информацию о процессе загрузки и принимают команды системного управления через последовательный порт. Этот порт и параметр baudrate, используемые в сочетании с EMS, указываются в строках redirect= и redirectbaudrate= в разделе [boot loader] файла Boot.ini
/SAFEBOOT:	Задает параметры загрузки в безопасном режиме. Никогда не указывайте этот параметр вручную, так как Ntldr сам задает его при использовании меню, открываемого нажатием клавиши F8. (При загрузке Windows в безопасном режиме загружаются только драйверы и сервисы, указанные в подразделах реестра Minimal или Network, которые находятся в HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot.) После двоеточия можно указать один из трех дополнительных параметров: MINIMAL, NETWORK или DSREPAIR. Параметр MINIMAL соответствует безопасной загрузке без сетевой поддержки, NETWORK — безопасной загрузке с сетевой поддержкой. DSREPAIR (Directory Services Repair) предназначен для такой загрузки Windows, при которой Active Directory находится в автономном состоянии (offline), а база данных этой службы не открывается. Это позволяет администратору выполнять диагностику, исправлять или восстанавливать базу данных. Можно использовать еще один дополнительный параметр, ALTERNATESHELL, — тогда вместо оболочки по умолчанию (Windows Explorer) загружается то, что указано в разделе реестра HKLM\System\CurrentControlSet\SafeBoot\AlternateShell

см. след. стр.

Таблица 5-2. (*окончание*)

Параметр	Описание
/SCSIORDINAL:	Указывает идентификатор SCSI-контроллера для Windows явным образом. (Добавление нового SCSI-устройства в систему с интегрированным SCSI-контроллером может привести к смене идентификатора этого контроллера.) Подробности см. в статье Q103625 в Microsoft Knowledge Base
/SDIBOOT=	Применяется в системах Windows XP Embedded, чтобы Windows загружалась с образа RAM-диска, хранящегося в указанном файле System Disk Image (SDI)
/SOS	Заставляет Windows перечислять на экране список помеченных к загрузке драйверов устройств, а затем показывать номер версии системы (включая номер сборки), объем физической памяти и число процессоров
/TIMERES=	В системах со стандартной многопроцессорной x86-версией HAL (Halmps.dll) задает разрешение системного таймера. Аргументом является значение в сотнях наносекунд, но частота устанавливается в соответствии с ближайшим меньшим значением, поддерживаемым HAL (см. ниже). Сотни наносекунд — Миллисекунды (мс) 97660,98 — 195322,00 390633,90 — 781257,80 Разрешение по умолчанию — 7,8 мс. Разрешение системного таймера влияет на разрешение ожидаемых таймеров. Например, параметр /TIMERES=21000 установит разрешение таймера равным 2,0 мс
/USERVA= (только для 32-разрядных систем)	Поддерживается только в Windows XP и Windows Server 2003. Как и параметр /3GB, предоставляет приложениям большее адресное пространство. Указывается в мегабайтах, значение должно укладываться в диапазон от 2048 до 3072. Этот параметр предъявляет к приложениям те же требования, что и параметр /3GB; используется только в сочетании с /3GB
/WIN95	Указывает Ntldr загрузить содержимое файла Bootsect.w40, в котором хранится загрузочный сектор Windows 9*x*. Применим, только если на компьютере установлены три операционные системы (Windows, Windows 9*x* и MS-DOS). Подробности см. в статье Q157992 в Microsoft Knowledge Base
/WIN95DOS	Указывает Ntldr использовать загрузочный сектор MS-DOS, хранящийся в файле Bootsect.w40. Применим, только если на компьютере установлены три операционные системы (Windows, Windows 9*x* и MS-DOS). Подробности см. в статье Q157992 в Microsoft Knowledge Base
/YEAR=	Указывает Windows игнорировать значение года, сообщаемое системными часами компьютера, и использовать заданное значение. Этот параметр влияет на все программное обеспечение, в том числе на ядро Windows. Пример: /YEAR=2001. (Этот параметр предназначался для тестирования на совместимость с датами 2000 года)

Если до истечения периода ожидания, указанного в Boot.ini, пользователь не выбрал ни одной команды загрузочного меню, Ntldr выбирает вариант по умолчанию, который соответствует самой верхней записи в Boot.ini и содержит путь, совпадающий с путем в строке «default=». После выбора одного из вариантов Ntldr загружает и запускает Ntdetect.com, 16-разрядную программу реального режима, которая получает от BIOS сведения о базовых устройствах и конфигурации компьютера:

- время и дату, хранящиеся в энергонезависимой памяти CMOS;
- типы шин в системе (например, ISA, PCI, EISA, MCA) и идентификаторы устройств, подключенных к этим шинам;
- число, емкость и тип дисков, присутствующих в системе;
- тип подключенной мыши;
- число и тип параллельных портов, сконфигурированных в системе;
- типы видеоадаптеров, присутствующих в системе.

Эти сведения, записываемые во внутренние структуры данных, на более поздних этапах загрузки будут сохранены в разделе реестра HKLM\HARDWARE\DESCRIPTION.

Далее Ntldr в Windows 2000 очищает экран и выводит индикатор прогресса загрузки с надписью «Starting Windows» («Запуск Windows»). Индикатор остается на нулевой отметке до начала загрузки драйверов устройств (см. п. 5 в следующем списке). Под индикатором появляется сообщение: «For troubleshooting and advanced startup options for Windows 2000, press F8» («Для выбора особых вариантов загрузки Windows 2000 нажмите F8»). При нажатии клавиши F8 выводится дополнительное загрузочное меню, предлагающее выбрать особые варианты загрузки — последнюю удачную конфигурацию, безопасный или отладочный режим и т. д. В Windows XP и Windows Server 2003 Ntldr выводит экран-заставку вместо индикатора прогресса загрузки.

Если Ntldr выполняется в x64-системе и в загрузочном меню выбран элемент, указывающий на запуск ядра для x64, то Ntldr переключает процессор в режим, в котором «родной» размер слова составляет 64 бита. Затем Ntldr начинает загружать необходимые для инициализации ядра файлы. Загрузочным является том, соответствующий разделу, на котором находится системный каталог (обычно \Windows) загружаемой системы. Ниже описываются операции, выполняемые Ntldr.

1. Загружает соответствующие образы ядра и HAL (по умолчанию — Ntoskrnl.exe и Hal.dll). Если Ntldr не удается загрузить какой-либо из этих файлов, он выводит сообщение «Windows could not start because the following file was missing or corrupt» («Не удается запустить Windows из-за испорченного или отсутствующего файла»), за которым следует имя файла.
2. Для поиска драйверов устройств, которые нужно загрузить, считывает в память содержимое куста реестра SYSTEM, \Windows\System32\Config\System. Куст — это файл, содержащий поддерево реестра. Подробнее о реестре см. главу 4.

3. Сканирует загруженный в память куст реестра SYSTEM и находит все загрузочные драйверы устройств (это драйверы, обязательные для запуска системы). Они отмечены в реестре флагом SERVICE_BOOT_START (0). Каждому драйверу устройства в реестре соответствует подраздел HKLM\SYSTEM\CurrentControlSet\Services. Например, драйверу диспетчера логических дисков (Logical Disk Manager) в разделе Services соответствует подраздел Dmio, показанный на рис. 5-2. (Подробное описание содержимого Services см. в разделе «Сервисы» главы 4.)

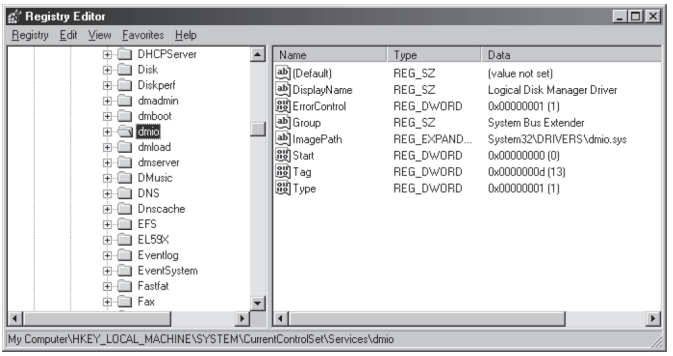


Рис. 5-2. *Параметры драйвера диспетчера логических дисков*

4. Вносит в список загрузочных драйверов устройств драйвер файловой системы, отвечающий за реализацию кода для конкретного типа раздела (FAT, FAT32 или NTFS), на котором находится системный каталог. Ntldr должен загрузить этот драйвер именно сейчас, иначе ядро будет требовать от драйверов их же загрузки, и получится замкнутый круг.
5. Загружает драйверы, обязательные для запуска системы. Ход загрузки отражается индикатором «Starting Windows». Полоска на индикаторе продвигается вперед по мере загрузки драйверов (число загрузочных драйверов считается равным 80, поэтому после успешной загрузки каждого драйвера полоска продвигается на 1,25%). Если в Boot.ini указан параметр /SOS, то вместо индикатора Ntldr выводит имя файла каждого загрузочного драйвера. Учтите, что на этом этапе драйверы лишь загружаются, а их инициализация происходит позже.
6. Подготавливает регистры процессора для выполнения Ntoskrnl.exe.

На этом участие Ntldr в процессе загрузки заканчивается. Для инициализации системы Ntldr вызывает главную функцию из Ntoskrnl.exe.

Процесс загрузки на платформе IA64

Файлы, участвующие в процессе загрузки на платформе IA64 перечислены в таблице 5-3. Системы IA64 соответствуют спецификации Extensible Firmware Interface (EFI), определенной Intel. В EFI-совместимой системе имеется микрокод, который запускает стартовый загрузчик (загрузочный код), записываемый Windows Setup в системную NVRAM (nonvolatile RAM). Загру-

зочный код считывает содержимое IA64-эквивалента Boot.ini, который также хранится в NVRAM. Средства Microsoft EFI можно запускать в консоли EFI, а Bootcfg.exe (утилита, поставляемая с Windows) позволяет модифицировать параметры и варианты загрузки из NVRAM.

Далее происходит распознавание оборудования, в ходе которого стартовый загрузчик использует интерфейсы EFI для определения числа и типов следующих устройств:

- сетевых адаптеров;
- видеоадаптеров;
- клавиатур;
- дисковых контроллеров;
- накопителей.

Так же, как и Ntldr в системах x86 и x64, стартовый загрузчик выводит меню с вариантами загрузки. Как только пользователь выбирает один из вариантов, загрузчик переходит в подкаталог в разделе EFI System, соответствующий выбранному варианту, и загружает несколько других файлов, необходимых для продолжения загрузки: Fpswa.efi и Ia64ldr.efi. Спецификация EFI требует, чтобы в системе был раздел, обозначенный как EFI System; он форматируется под файловую систему FAT и может быть размером от 100 Мб до 1 Гб (или до 1% от общего размера диска). Для каждой установленной Windows-системы выделяется свой подкаталог в разделе EFI System (в каталоге EFI\Microsoft). Первой системе назначается подкаталог Winnt50, второй — Winnt50.1 и т. д. Ia64ldr.efi отвечает за загрузку Ntoskrnl.exe, Hal.dll и драйверов, применяемых на этапе загрузки. Далее процесс загрузки идет так же, как и на платформе x86 или x64.

Таблица 5-3. *Компоненты, участвующие в процессе загрузки на платформе IA64*

Компонент	Местонахождение	Описание
Fpswa.efi	EFI\Microsoft\Winnt50.x в разделе EFI System	Файл с кодом поддержки операций с плавающей точкой
Ia64ldr.efi	EFI\Microsoft\Winnt50.x в разделе EFI System	Загружает Ntoskrnl.exe, Hal.dll и драйверы, используемые на этапе загрузки
Ntoskrnl.exe	\Windows\System32	Инициализирует подсистемы исполнительной системы, а также драйверы устройств, используемые на этапе загрузки и при запуске системы (boot and system-start device drivers), подготавливает систему к запуску встроенных приложений и запускает Smss.exe
Hal.dll	\Windows\System32	DLL режима ядра, предоставляющая интерфейс к оборудованию для Ntoskrnl и драйверов
Диспетчер управления сервисами (SCM)	\Windows\System32	Загружает и инициализирует автоматически запускаемые драйверы устройств и сервисы Windows

см. след. стр.

Таблица 5-3. (*окончание*)

Компонент	Местонахождение	Описание
Smss	\Windows\System32	Загружает подсистему Windows, в том числе Win32k.sys и Csrss.exe, а затем запускает процесс Winlogon
Winlogon	\Windows\System32	Запускает диспетчер управления сервисами (SCM), подсистему Local Security Authority Subsystem (LSASS) и выводит диалоговое окно для интерактивного входа в систему

Инициализация ядра и компонентов исполнительной системы

Вызывая Ntoskrnl.exe, Ntldr передает структуру данных с копией строки из Boot.ini (представляющей выбранный вариант загрузки), с указателем на таблицы памяти (сгенерированные Ntldr для описания физической памяти в данной системе), с указателем на загруженные в память копии кустов реестра HARDWARE и SYSTEM и с указателем на список загруженных драйверов.

Ntoskrnl начинает первую из двух фаз процесса инициализации (они нумеруются от 0). Большинство компонентов исполнительной системы имеет инициализирующую функцию, которая принимает параметр, определяющий текущую фазу.

В фазе 0 прерывания отключены. Предназначение этой фазы в том, чтобы сформировать рудиментарные структуры, необходимые для вызова сервисов в фазе 1. Главная функция Ntoskrnl вызывает *KiSystemStartup*, которая в свою очередь вызывает *HalInitializeProcessor* и *KiInitializeKernel* для каждого процессора. Работая на стартовом процессоре, *KiInitializeKernel* выполняет общесистемную инициализацию ядра, в том числе всех внутренних структур данных, разделяемых всеми процессорами. Затем каждый экземпляр *KiInitializeKernel* вызывает функцию *ExpInitializeExecutive*, отвечающую за управление фазой 0.

ExpInitializeExecutive начинает с вызова HAL-функции *HalInitSystem*, позволяющей HAL взять управление инициализацией системы на себя. Одной из задач *HalInitSystem* является подготовка системного контроллера прерываний каждого процессора к обработке прерываний и конфигурирование таймера, используемого для учета распределяемого процессорного времени (подробнее на эту тему см. раздел «Учет квантов времени» главы 6).

Лишь на стартовом процессоре *ExpInitializeExecutive* не просто вызывает *HalInitSystem*, но и выполняет другие операции по инициализации. Когда *HalInitSystem* возвращает управление, функция *ExpInitializeExecutive*, выполняемая на стартовом процессоре, обрабатывает параметр /BURNMEMORY файла Boot.ini (если таковой указан и действителен для данного варианта загрузки). В соответствии с этим параметром *ExpInitializeExecutive* исключает указанный объем памяти.

Далее *ExpInitializeExecutive* вызывает процедуры инициализации для диспетчера памяти, диспетчера объектов, монитора состояния защиты, диспетчера процессов и диспетчера Plug and Play. Эти компоненты выполняют следующие инициализирующие операции.

1. Диспетчер памяти формирует таблицы страниц и внутренние структуры данных, необходимые для предоставления базовых сервисов, связанных с памятью. Кроме того, он создает и резервирует пространство для кэша файловой системы, а также выделяет области для пулов подкачиваемой и неподкачиваемой памяти. Другие компоненты исполнительной системы, ядро и драйверы устройств пользуются этими пулами, выделяя память под собственные структуры данных.
2. При инициализации диспетчера объектов определяются объекты, необходимые для создания его пространства имен, чтобы другие компоненты могли помещать в него свои объекты. Также создается таблица описателей для поддержки учета ресурсов.
3. Монитор состояния защиты инициализирует объект типа «маркер доступа» и использует его для создания и подготовки первого маркера по учетной записи локальной системы, назначаемого начальному процессу (об учетной записи локальной системы см. главу 8).
4. Диспетчер процессов производит бóльшую часть своей инициализации в фазе 0, определяя типы объектов «процесс» и «поток» и создавая списки для отслеживания активных процессов и потоков. Он также создает объект «процесс» для начального процесса и присваивает ему имя Idle. Наконец, диспетчер процессов создает процесс System и системный поток для выполнения процедуры *Phase1Initialization*. Этот поток не запускается сразу после создания, поскольку прерывания пока запрещены.
5. Далее наступает фаза 0 в инициализации диспетчера Plug and Play, в ходе которой просто инициализируется ресурс исполнительной системы, используемый для синхронизации ресурсов шин.

Когда на каждом процессоре управление возвращается к функции *KiInitializeKernel*, она передает его циклу Idle. В результате системный поток, созданный, как было описано в п. 4 предыдущего списка, начинает фазу 1. Дополнительные процессоры ждут начала своей инициализации до п. 5 фазы 1 (см. список ниже). В фазе 1 выполняются следующие операции. (На экранезаставке, выводимом при загрузке Windows 2000, отображается индикатор прогресса, поэтому в списке упоминаются операции, связанные с обновлением этого индикатора.)

1. Для подготовки системы к приему прерываний от устройств и для разрешения прерываний вызывается *HalInitSystem*.
2. Вызывается загрузочный видеодрайвер (\Windows\System32\Bootvid.dll), который выводит экран-заставку, показываемую в процессе запуска Windows. (В Windows XP и Windows Server 2003 этот драйвер отображает ту картинку, которую Ntldr ранее вывел на экран.)

3. Инициализируется диспетчер электропитания.
4. Инициализируются системные часы (вызовом *HalQueryRealTimeClock*), текущее значение которых сохраняется как время загрузки системы.
5. В многопроцессорной системе инициализируются остальные процессоры и начинается выполнение команд.
6. Индикатор прогресса загрузки устанавливается на отметку 5%.
7. Диспетчер объектов создает корневой каталог пространства имен (\), каталог \ObjectTypes и каталог сопоставлений DOS-имен устройств (\?? в Windows 2000 или \Global?? в Windows XP и Windows Server 2003), а также символьную ссылку в каталоге сопоставлений DOS-имен устройств.
8. Вызывается исполнительная система для создания своих типов объектов, включая семафор, мьютекс, событие и таймер.
9. Ядро инициализирует структуры данных планировщика (диспетчера) и таблицу диспетчеризации системных сервисов.
10. Монитор состояния защиты создает в пространстве имен диспетчера объектов каталог \Security и инициализирует структуры данных аудита (если аудит системы разрешен).
11. Индикатор прогресса загрузки устанавливается на отметку 10%.
12. Для создания объекта «раздел» и системных рабочих потоков вызывается диспетчер памяти (см. главу 7).
13. На системное адресное пространство проецируются таблицы NLS (National Language Support).
14. На системное адресное пространство проецируется Ntdll.dll.
15. Диспетчер кэша инициализирует структуры данных кэша файловой системы и создает свои рабочие потоки.
16. Диспетчер конфигурации создает в пространстве имен объект «раздел реестра» \Registry и копирует переданные Ntldr начальные данные в кусты реестра HARDWARE и SYSTEM.
17. Инициализируются структуры данных драйвера файловой системы.
18. Диспетчер Plug and Play вызывает PnP BIOS.
19. Индикатор прогресса загрузки устанавливается на отметку 20%.
20. Подсистема LPC инициализирует объект типа «порт LPC».
21. Если система запущена с протоколированием загрузки (/BOOTLOG), инициализируется файл протокола загрузки.
22. Индикатор прогресса загрузки устанавливается на отметку 25%.
23. Наступает момент инициализации диспетчера ввода-вывода. Согласно показаниям индикатора, эта стадия запуска системы занимает 50% времени. После успешной загрузки каждого драйвера диспетчер ввода-вывода продвигает полоску на индикаторе на 2% (если загружается более 25 драйверов, индикатор останавливается на отметке 75%).

Диспетчер ввода-вывода прежде всего инициализирует различные внутренние структуры и создает типы объектов «устройство» и «драйвер». Затем он вызывает диспетчер Plug and Play, диспетчер электропитания и HAL, чтобы начать динамическое перечисление и инициализацию устройств. (Подробнее этот сложный процесс, специфичный для конкретной подсистемы ввода-вывода, рассматривается в главе 9.) Далее инициализируется подсистема WMI (Windows Management Instrumentation), которая предоставляет WMI-поддержку драйверам устройств. (Подробнее о WMI см. раздел «Windows Management Instrumentation» главы 4.) После этого вызываются все загрузочные драйверы, которые осуществляют свою инициализацию. Также загружаются и инициализируются драйверы, необходимые для запуска системы (см. главу 9). Наконец, в пространстве имен диспетчера объектов создаются имена устройств MS-DOS в виде символьных ссылок.

24. Индикатор прогресса загрузки устанавливается на отметку 75%.
25. Если система загружается в безопасном режиме, этот факт отмечается в реестре.
26. Включается подкачка страниц для кода режима ядра (в Ntkrnl и драйверах), если она явно не запрещена в реестре.
27. Индикатор прогресса загрузки устанавливается на отметку 80%.
28. Вызывается диспетчер электропитания для инициализации своих структур данных.
29. Индикатор прогресса загрузки устанавливается на отметку 85%.
30. Вызывается монитор состояния защиты для создания потока Command Server, взаимодействующего с LSASS (см. раздел «Компоненты системы защиты» главы 8).
31. Индикатор прогресса загрузки устанавливается на отметку 90%.
32. На завершающем этапе создается процесс Smss диспетчера сеансов (базовые сведения об Smss см. в главе 2). Smss отвечает за создание среды пользовательского режима, которая предоставляет визуальный интерфейс Windows. Об инициализации Smss см. следующий раздел.
33. Индикатор прогресса загрузки устанавливается на отметку 100%.

Перед завершением инициализации компонентов исполнительной системы и ядра поток инициализации фазы 1 в течение пяти секунд ждет освобождения описателя процесса Smss. Если этот процесс завершается до истечения пяти секунд, происходит крах системы с кодом SESSION5_INITIALIZATION_FAILED.

По истечении пяти секунд запуск диспетчера сеансов считается успешным, и вызывается функция потока обнуления страниц диспетчера памяти (см. главу 7).

Smss, Csrss и Winlogon

Smss похож на любой другой процесс пользовательского режима, но имеет два существенных отличия. Во-первых, Windows считает его *доверяемой* (trusted) частью системы. Во-вторых, Smss является *встроенным* (native) приложением. Как доверяемый компонент Smss может выполнять операции, доступные лишь немногим процессам, например создавать маркеры защиты. А как встроенное приложение Smss использует не Windows API, а базовые API-функции исполнительной системы, в совокупности называемые Windows Native API. Smss не обращается к Windows API, поскольку при его запуске подсистема Windows еще не функционирует. Запуск подсистемы Windows и является одной из его первых задач.

Затем Smss вызывает диспетчер конфигурации, который завершает инициализацию реестра, заполняя все его разделы. Диспетчер конфигурации запрограммирован так, что ему известно местонахождение всех кустов реестра на диске, кроме содержащих пользовательские параметры. Пути ко всем загружаемым им кустам реестра записываются в раздел HKLM\SYSTEM\CurrentControlSet\Control\Hivelist.

Основной поток Smss выполняет следующие инициализирующие операции.

1. Создает объект «порт LPC» (*\SmApiPort*) и два потока, ожидающие клиентские запросы (например, на загрузку новой подсистемы или на создание сеанса).
2. Определяет символьные ссылки на имена устройств MS-DOS (вроде COM1 и LPT1).
3. Если установлены Terminal Services, создает в пространстве имен диспетчера объектов каталог \Sessions (для нескольких сеансов).
4. Запускает программы, указанные в HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\BootExecute. Как правило, в нем содержится одна команда на запуск Autochk (версия Chkdsk, работающая на этапе загрузки).
5. Выполняет отложенные действия по переименованию и удалению файлов, указанные в разделах HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\PendingFileRenameOperations и HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\PendingFileRenameOperations2.
6. Открывает известные DLL и создает для них объекты «раздел» в каталоге \Knowndlls пространства имен диспетчера объектов. Список DLL, считаемых известными, находится в разделе HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs, а путь к каталогу, где расположены эти DLL, хранится в параметре Dlldirectory этого раздела. Об использовании разделов Known DLLs при загрузке DLL см. главу 6.
7. Создает дополнительные страничные файлы. Их конфигурация хранится в разделе HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagingFiles.
8. Инициализирует реестр. Диспетчер конфигурации заполняет реестр, загружая кусты HKLM\SAM, HKLM\SECURITY и HKLM\SOFTWARE. Хотя ин-

формация о местонахождении файлов кустов содержится в разделе HKLM\SYSTEM\CurrentControlSet\Control\Hivelist, диспетчер конфигурации ищет эти файлы в каталоге \Windows\System32\Config.

9. Создает системные переменные окружения, определенные в HKLM\System\CurrentControlSet\Session Manager\Environment.
10. Загружает часть подсистемы Windows, работающую в режиме ядра (Win32k.sys). Smss определяет местонахождение Win32k.sys и других загружаемых им компонентов по путям, хранящимся в HKLM\SYSTEM\CurrentControlSet\Control\Session Manager. Инициализирующий код в Win32k.sys использует видеодрайвер для переключения экрана в разрешение, определенное в профиле по умолчанию. Таким образом, в этот момент видеоадаптер переключается с VGA-режима, используемого загрузочным видеодрайвером, в выбранное для данной системы разрешение.
11. Запускает процессы подсистем, в том числе Csrss. (Как говорилось в главе 2, подсистемы POSIX и OS/2 в Windows 2000 запускаются по требованию.)
12. Запускает процесс Winlogon. Этапы запуска Winlogon кратко описываются ниже.
13. Создает порты LPC для сообщений об отладочных событиях (*DbgSsApiPort* и *DbgUiApiPort*) и потоки, прослушивающие эти порты.

Отложенные действия по переименованию файлов

Тот факт, что исполняемые образы и DLL при использовании проецируются в память, делает невозможным обновление базовых системных файлов по окончании загрузки Windows. API-функция *MoveFileEx* позволяет указать, что перемещение файла должно быть отложено до следующей загрузки. Пакеты обновлений и критические исправления, которым нужно обновлять уже используемые файлы, проецируемые в память, устанавливают заменяющие файлы во временные каталоги и вызывают функцию *MoveFileEx* именно так, как говорилось чуть выше. В этом случае *MoveFileEx* просто записывает команды в параметры *PendingFileRenameOperations* и *PendingFileRenameOperations2* в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager. Эти параметры имеют тип MULTI_SZ, и каждая операция указывается парами имен файлов: первое имя — источник, а второе — приемник. В операциях удаления вместо приемника задается пустая строка. Чтобы просмотреть зарегистрированные отложенные команды переименования и удаления, используйте утилиту Pendmoves с сайта *www.sysinternals.com*.

После выполнения вышеперечисленных операций основной поток Smss переходит к бесконечному ожиданию описателей процессов Csrss и Winlogon. Поскольку от этих процессов зависит функционирование Windows, в случае их неожиданного завершения Smss вызывает крах системы. (В Windows XP и выше, если Csrss по какой-то причине завершается, крах системы вызывается ядром, а не Smss.)

Далее Winlogon продолжает инициализацию, выполняя такие операции, как создание начального объекта WindowStation и объектов рабочего стола. Если в HKLM\Software\Microsoft\Windows NT\Current Version\WinLogon\GinaDLL указана какая-нибудь DLL, Winlogon использует ее в качестве GINA; в ином случае применяется GINA по умолчанию от Microsoft, Msgina (\Windows\System32\Msgina.dll), которая отображает стандартное диалоговое окно входа в Windows. Затем Winlogon создает процесс SCM (диспетчера управления сервисами) (\Windows\System32\Services.exe), который загружает все сервисы и драйверы устройств, помеченные для автоматического запуска, а также запускает процесс LSASS (подсистемы локальной аутентификации) (\Windows\System32\Lsass.exe). Подробнее о запуске Winlogon и LSASS см. раздел «Инициализация Winlogon» главы 8.

После того как SCM инициализирует автоматически запускаемые сервисы и драйверы устройств, а пользователь успешно зарегистрируется в системе, загрузка считается успешно завершенной. Параметры в разделе HKLM\SYSTEM\Select\LastKnownGood обновляются в соответствии со значениями параметров последней удачной конфигурации (\CurrentControlSet).

ПРИМЕЧАНИЕ Если на неинтерактивном сервере не бывает интерактивного входа, раздел LastKnownGood, отражающий набор управления (control set), который позволил выполнить успешную загрузку, не обновляется.

Вы можете заменить определение успешной загрузки. Для этого установите HKLM\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\ReportBootOk в 0, напишите свою программу, проверяющую успешность загрузки, и укажите путь к ней в HKLM\System\CurrentControlSet\Control\BootVerificationProgram. Такая программа должна вызывать API-функцию *NotifyBootConfigStatus*, если загрузка прошла успешно.

Запустив SCM, Winlogon ждет уведомления об интерактивном входе от GINA. Получив такое уведомление и проверив вход (об этом процессе см. в главе 8), Winlogon загружает куст реестра из профиля зарегистрировавшегося пользователя и отображает его на HKCU. Затем он настраивает переменные окружения для данного пользователя, хранящиеся в HKCU\Environment, и направляет уведомления о входе компонентам, зарегистрированным в HKLM\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Notify.

Затем Winlogon сообщает GINA запустить оболочку. В ответ на этот запрос Msgina запускает исполняемый файл (или исполняемые файлы), указанный в параметре HKLM\Software\Microsoft\Windows NT\CurrentVersion\WinLogon\Userinit (несколько исполняемых файлов перечисляются через запятые), который по умолчанию указывает на \Windows\System32\Userinit.exe. Userinit.exe выполняет следующие операции.

1. Обрабатывает пользовательские сценарии, указанные в HKCU\Software\Policies\Microsoft\Windows\System\Scripts, и машинные сценарии входа, заданные в HKLM\Software\Policies\Microsoft\Windows\System\Scripts.

(Так как машинные сценарии выполняются после пользовательских, они могут переопределять пользовательские настройки.)

2. Если политика группы задает какую-либо квоту в профиле пользователя, Userinit.exe запускает \Windows\System32\Proquota.exe для ее применения.
3. Запускает оболочку (или оболочки), указанную в HKCU\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell. Если этого параметра нет, Userinit.exe запускает оболочку (или оболочки), определенные в HKLM\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell (по умолчанию — Explorer.exe).

Далее Winlogon уведомляет зарегистрированные провайдеры сетей о входе пользователя. Провайдер сети Microsoft, маршрутизатор многосетевого доступа (Multiple Provider Router, MPR) (\Windows\System32\Mpr.dll), восстанавливает постоянные подключения к сетевому диску и принтерам, установленные пользователем; эти сопоставления хранятся в HCU\Network и HKCU\Printers соответственно. На рис. 5-3 показано дерево процессов, которое отображается в Process Explorer при входе до завершения Userinit.

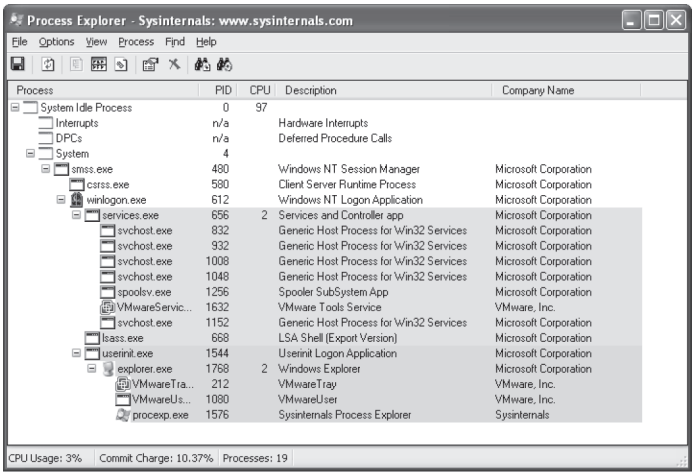


Рис. 5-3. *Дерево процессов при входе*

Автоматически запускаемые образы

Помимо параметров Userinit и Shell в разделе Winlogon, существует много других разделов в реестре и каталогов, проверяемых и обрабатываемых системными компонентами для автоматического запуска процессов при загрузке и входе. Утилита Msconfig (в Windows XP и Windows Server 2003 это \Windows\System32\Msconfig.exe) показывает образы, сконфигурированные в нескольких местах. Но утилита Autoruns от Sysinternals (*www.sysinternals.com*), представленная на рис. 5-4, анализирует больше разделов реестра и каталогов, чем Msconfig, и выводит больше информации об образах, настроенных на автоматический запуск. По умолчанию Autoruns показывает толь-

ко те места, где задается автоматический запуск хотя бы одного образа, но команда Include Empty Locations в меню View заставляет Autoruns отображать все проверяемые ею разделы реестра и каталоги. В меню View можно настроить Autoruns на отображение сведений о других типах автоматически запускаемых образов, например служб Windows и надстроек Explorer.

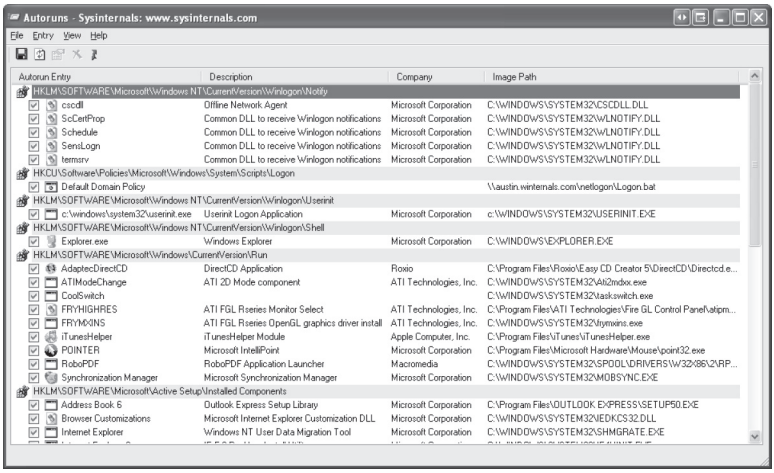


Рис. 5-4. *Утилита Autoruns от* www.sysinternals.com

ЭКСПЕРИМЕНТ: утилита Autoruns

Многие пользователи даже не представляют, сколько программ выполняется в процессе их входа. OEM (original equipment manufacturers) часто конфигурируют свои системы с помощью дополнительных утилит, которые выполняются в фоновом режиме и обычно не видны. Чтобы увидеть, какие программы настроены на автоматический запуск на вашем компьютере, запустите утилиту Autoruns; ее можно скачать с сайта *www.sysinternals.com*. Сравните полученный в Autoruns список с тем, что показывается Msconfig (доступной в Windows XP и Windows Server 2003), и обратите внимание на различия. Потом попробуйте разобраться в предназначении каждой программы.

Анализ проблем при загрузке и запуске системы

В этом разделе представлены подходы к решению проблем, возможных в процессе запуска Windows из-за повреждения жесткого диска и файлов, отсутствия каких-либо файлов и ошибок в сторонних драйверах. Сначала мы опишем три режима восстановления Windows при возникновении проблем с загрузкой: последняя удачная конфигурация, безопасный режим и консоль восстановления (Recovery Console). Затем мы расскажем о наиболее распространенных проблемах при загрузке, об их причинах и способах устранения.

Последняя удачная конфигурация

Последняя удачная конфигурация (last known good, LKG) — полезный механизм для возврата системы, рухнувшей в процессе загрузки, в загружаемое состояние. Поскольку параметры системной конфигурации хранятся в HKLM\System\CurrentControlSet\Control, конфигурация драйверов и сервисов — в HKLM\System\CurrentControlSet\Services, изменения этих частей реестра могут привести к тому, что система станет незагружаемой. Например, если вы установили драйвер устройства с ошибкой, из-за которой происходит крах системы при загрузке, то можете нажать клавишу F8 в момент загрузки и выбрать из меню последнюю удачную конфигурацию. Система отмечает набор управления, использовавшийся при загрузке как неудачный, устанавливая параметр Failed в HKLM\System\Select и заменяя значение параметра HKLM\System\Select\Current на значение параметра HKLM\System\Select\LastKnownGood. Она также обновляет символьную ссылку HKLM\System\CurrentControlSet так, чтобы она указывала на набор управления LastKnownGood. Поскольку для нового драйвера нет подраздела в разделе Services набора управления LastKnownGood, система успешно загрузится.

Безопасный режим

Наиболее распространенная причина, по которой системы Windows становятся незагружаемыми, заключается в том, что какой-то драйвер устройства приводит к краху при загрузке. Поскольку со временем программно-аппаратная конфигурация системы может измениться, скрытые до этого ошибки в драйверах могут проявиться в любой момент. Windows предоставляет администратору способ решения подобных проблем: загрузку в *безопасном режиме* (safe mode). Понятие безопасного режима в Windows заимствовано из потребительских версий Windows и представляет собой конфигурацию с минимальным набором драйверов и сервисов. Используя только самые необходимые драйверы, Windows избегает загрузки сторонних драйверов, способных вызывать крах системы.

Нажав клавишу F8 в начале загрузки Windows 2000, вы открываете дополнительное загрузочное меню, в котором присутствуют три варианта загрузки в безопасном режиме: Safe Mode (Безопасный режим), Safe Mode With Networking (Безопасный режим с загрузкой сетевых драйверов) и Safe Mode With Command Prompt (Безопасный режим с поддержкой командной строки). Стандартный безопасный режим подразумевает использование минимума необходимых для успешной загрузки драйверов. В безопасном режиме с сетевой поддержкой дополнительно загружаются сетевые драйверы и сервисы. Наконец, единственное отличие безопасного режима с поддержкой командной строки от стандартного заключается в том, что Windows вместо обычной оболочки Windows Explorer, позволяющей работать в GUI-режиме, запускает оболочку командной строки (Cmd.exe).

В Windows предусмотрен и четвертый безопасный режим — Directory Services Restore (Восстановление службы каталогов), который применяется

для загрузки системы с отключенной службой каталогов Active Directory и без открытия ее базы данных. Это позволяет администратору исправить поврежденную базу данных или восстановить ее с резервной копии. В этом режиме загружается весь набор драйверов и сервисов, кроме Active Directory. В тех случаях, в которых вам не удается войти в систему из-за повреждения базы данных Active Directory, этот режим дает возможность устранить неполадки.

Загрузка драйверов в безопасном режиме

Как Windows определяет набор драйверов для загрузки в стандартном безопасном режиме и безопасном режиме с сетевой поддержкой? Ответ следует искать в содержимом раздела реестра HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot. В нем присутствуют подразделы Minimal и Network. Каждый подраздел в свою очередь содержит подразделы с именами драйверов, сервисов или их групп. Так, подраздел vga.sys определяет драйвер видеоадаптера VGA, который поддерживает базовый набор графических сервисов стандартного видеоадаптера для IBM-совместимого компьютера. Этот драйвер используется системой в безопасном режиме вместо драйверов, которые позволяют задействовать все преимущества куда более совершенных видеоадаптеров, но способны помешать успешной загрузке системы. Каждому подразделу в разделе SafeBoot соответствует параметр по умолчанию, описывающий, что именно идентифицирует данный подраздел. Например, в подразделе vga.sys параметр по умолчанию — Driver.

Параметром по умолчанию для подраздела файловой системы является Driver Group. При создании сценария установки для драйвера устройства разработчик может указать, что он относится к какой-либо группе драйверов. Группы драйверов, определенные в системе, перечисляются в параметре List раздела реестра HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder. Разработчик приписывает драйвер к той или иной группе, чтобы указать Windows, на каком этапе загрузки следует запускать данный драйвер. Главное предназначение раздела ServiceGroupOrder — определение порядка загрузки групп драйверов. Некоторые группы драйверов нужно загружать до или после других. Параметр Group в подразделе реестра со сведениями о конфигурации драйвера, сопоставляет его с определенной группой. Подразделы со сведениями о конфигурации драйверов и сервисов находятся в разделе HKLM\SYSTEM\CurrentControlSet\Services. Взглянув на его содержимое, вы найдете раздел VgaSave для драйвера видеоадаптера VGA, который принадлежит к группе Video Save. Любой драйвер файловой системы, необходимый Windows для обращения к системному диску, находится в группе Boot File System. Если файловой системой такого диска является NTFS, то в группу входит драйвер NTFS; в ином случае в группу входит драйвер Fastfat (поддерживающий диски FAT12, FAT16 и FAT32). Другие драйверы файловой системы входят в группу File System, которая также включена в конфигурации Safe Mode и Safe Mode With Networking.

При загрузке в безопасном режиме Ntldr передает ядру (Ntoskrnl.exe) вместе с другими параметрами, указанными в Boot.ini для текущего варианта

загрузки, параметр командной строки /SAFEBOOT:, добавляя к нему одну или несколько строк в зависимости от выбранного типа безопасного режима. Для стандартного безопасного режима Ntldr добавляет MINIMAL, для Safe Mode With Networking — NETWORK, для Safe Mode With Command Prompt — MINIMAL(ALTERNATESHELL), а для Directory Services Restore — DSREPAIR.

Ядро Windows сканирует параметры загрузки в поисках спецификаторов безопасного режима и устанавливает значение внутренней переменной *InitSafeBootMode* в соответствии с результатом поиска. Значение этой переменной также записывается в раздел HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Option\OptionValue, что позволяет компонентам пользовательского режима (например, SCM) определять режим загрузки системы. Кроме того, при выборе Safe Mode With Command Prompt, ядро присваивает значение 1 параметру UseAlternateShell в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Option. Кроме того, ядро записывает параметры, переданные Ntldr, в раздел HKLM\SYSTEM\CurrentControlSet\Control\SystemStartOptions.

Когда диспетчер ввода-вывода загружает драйверы устройств, указанные в разделе HKLM\SYSTEM\CurrentControlSet\Services, он выполняет функцию *IopLoadDriver*. А когда диспетчер Plug and Play обнаруживает новое устройство и хочет динамически загрузить драйвер для этого устройства, он вызывает функцию *IopCallDriverAddDevice*. Обе эти функции перед загрузкой драйвера обращаются к функции *IopSafeBootDriverLoad*. Последняя проверяет значение переменной *InitSafeBootMode* и определяет, можно ли загрузить данный драйвер. Так, если система загружается в стандартном безопасном режиме, *IopSafeBootDriverLoad* ищет группу этого драйвера (если таковая есть) в подразделе Minimal. Найдя ее, *IopSafeBootDriverLoad* уведомляет вызвавшую функцию о том, что этот драйвер можно загрузить. В ином случае *IopSafeBootDriverLoad* ищет в том же подразделе имя драйвера. Если оно есть в списке, драйвер может быть загружен. Если *IopSafeBootDriverLoad* не находит в списке группу или имя данного драйвера, его загрузка запрещается. При загрузке системы в безопасном режиме с сетевой поддержкой *IopSafeBootDriverLoad* ведет поиск в подразделе Network, а в случае загрузки системы в нормальном режиме *IopSafeBootDriverLoad* разрешает загрузку всех драйверов.

Однако Ntldr загружает все драйверы, у которых в соответствующих разделах реестра значение Start равно 0, что указывает на необходимость их загрузки при запуске системы. Поскольку Ntldr не проверяет раздел SafeBoot (считая, что любой драйвер с нулевым значением параметра Start необходим для успешного старта системы), он загружает все загрузочные драйверы, которые впоследствии запускаются Ntoskrnl.

Программное обеспечение с поддержкой безопасного режима

SCM (Services.exe), проводя инициализацию при загрузке, проверяет параметр OptionValue в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Option, чтобы выяснить, загружается ли система в безопасном

режиме. Если да, SCM зеркально воспроизводит действия *IopSafeBootDriverLoad*. Он обрабатывает все сервисы, перечисленные в HKLM\SYSTEM\CurrentControlSet\Services, но загружает лишь отмеченные в соответствующем подразделе реестра для загрузки в безопасном режиме. Подробнее об инициализации SCM см. раздел «Сервисы» главы 4.

Userinit (\Windows\System32\Userinit.exe) — другой компонент пользовательского режима, которому нужно знать, загружается ли система в безопасном режиме. Userinit, инициализирующий среду для пользователя при его входе в систему, проверяет значение HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\UseAlternateValue. Если это значение установлено, в качестве пользовательской оболочки он запускает не Explorer.exe, а программу, указанную в HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\AlternateShell. Когда вы устанавливаете Windows на компьютер, параметру AlternateShell присваивается значение Cmd.exe, и командная строка Windows становится оболочкой по умолчанию для безопасного режима с командной строкой. Но, даже если текущей оболочкой является командная строка, из нее можно запустить Windows Explorer, введя команду *Explorer.exe*. Аналогичным образом из командной строки можно запустить любую GUI-программу.

А как приложения узнают о загрузке системы в безопасном режиме? Вызовом Windows-функции *GetSystemMetrics(SM_CLEANBOOT)*. Пакетные сценарии, выполняющие некоторые действия при загрузке системы в безопасном режиме, проверяют наличие переменной окружения SAFEBOOT_OPTION, так как система определяет ее только при загрузке в безопасном режиме.

Ведение протокола при загрузке в безопасном режиме

Если вы загружаете систему в безопасном режиме, Ntldr передает ядру Windows вместе с параметрами, устанавливающими безопасный режим, и параметр /BOOTLOG. При инициализации ядро проверяет наличие параметра /BOOTLOG независимо от того, задан ли безопасный режим. Если ядро обнаруживает соответствующую строку, оно протоколирует все свои действия при загрузке каждого драйвера. Так, если функция *IopSafeBootDriverLoad* запрещает загрузку какого-либо драйвера, диспетчер ввода-вывода вызывает функцию *IopBootLog*, регистрируя, что данный драйвер не загружен. Аналогичным образом после успешной загрузки драйвера, входящего в конфигурацию безопасного режима, *IopLoadDriver* вызывает *IopBootLog* для внесения записи о загрузке этого драйвера. Изучив файлы протокола загрузки, можно выяснить, какие драйверы являются частью данной конфигурации.

Поскольку ядро избегает изменения данных на диске до запуска Chkdsk, который происходит на более позднем этапе загрузки, *IopBootLog* не может просто сбрасывать записи в файл. Вместо этого она записывает их в раздел реестра HKLM\SYSTEM\CurrentControlSet\BootLog. Диспетчер сеансов (Smss), первый загружаемый компонент пользовательского режима, запускает Chkdsk для проверки целостности системного диска, а потом завершает инициализацию реестра, вызывая *NtInitializeRegistry*. Этот вызов указывает ядру, что уже можно безопасно открыть на диске файл протокола, что и делается вызовом

IopCopyBootLogRegistryToFile. Эта функция создает в системном каталоге Windows (по умолчанию — \Windows) файл Ntbtlog.txt и копирует в него содержимое раздела реестра BootLog. *IopCopyBootLogRegistryToFile* также устанавливает флаг, сигнализирующий *IopBootLog* о возможности записи непосредственно в файл протокола. Ниже показан фрагмент образца такого файла.

```
Service Pack 1 3 30 2004 14:05:21.500
Loaded driver \WINDOWS\system32\ntoskrnl.exe
Loaded driver \WINDOWS\system32\hal.dll
Loaded driver \WINDOWS\system32\KDCOM.DLL
Loaded driver \WINDOWS\system32\BOOTVID.dll
Loaded driver ACPI.sys
Loaded driver \WINDOWS\System32\DRIVERS\WMILIB.SYS
Loaded driver pci.sys
Loaded driver isapnp.sys
Loaded driver intelide.sys
Loaded driver \WINDOWS\System32\DRIVERS\PCIIDEX.SYS
Loaded driver MountMgr.sys
Loaded driver ftdisk.sys
Loaded driver dmload.sys
Loaded driver dmio.sys Microsoft (R)Windows 2000 (R)Version 5.0
 (Build 2195) 2 11 2000 10:53:27.500
Loaded driver \WINNT \System32 \ntoskrnl.exe
Loaded driver \WINNT \System32 \hal.dll
Loaded driver \WINNT \System32 \BOOTVID.DLL
Loaded driver ACPI.sys
Loaded driver \WINNT \System32 \DRIVERS \WMILIB.SYS
Loaded driver pci.sys
Loaded driver isapnp.sys
Loaded driver compbatt.sys
Loaded driver \WINNT \System32 \DRIVERS \BATTC.SYS
Loaded driver intelide.sys
Loaded driver \WINNT \System32 \DRIVERS \PCIIDEX.SYS
Loaded driver pcmcia.sys
Loaded driver ftdisk.sys
Loaded driver Diskperf.sys
Loaded driver dmload.sys
Loaded driver dmio.sys
...
Did not load driver \SystemRoot\System32\Drivers\lbrtfdc.SYS
Did not load driver \SystemRoot\System32\Drivers\Sfloppy.SYS
Did not load driver \SystemRoot\System32\Drivers\i2omgmt.SYS
Did not load driver Media Control Devices
Did not load driver Communications Port
Did not load driver Audio Codecs
...
```

Консоль восстановления

В безопасном режиме обычно удается восстановить систему, ставшую незагружаемой в нормальном режиме из-за неправильной работы какого-либо драйвера устройства. Однако в некоторых ситуациях это не помогает: система все равно не загружается. Так, если сбойный драйвер входит в группу Safe, загрузиться в безопасном режиме не удастся. Другая ситуация, когда загрузка в безопасном режиме не помогает, — сбои в загрузочном драйвере стороннего поставщика, например в драйвере антивирусного сканера, поскольку загрузочные драйверы стартуют независимо от режима загрузки системы. Аналогичная ситуация возникает при повреждении файлов системных модулей или драйверов, входящих в конфигурацию безопасного режима, а также главной загрузочной записи (MBR) системного диска. Эти проблемы можно решить с помощью Recovery Console (Консоль восстановления). Консоль восстановления позволяет загрузить компактную оболочку командной строки с дистрибутивного компакт-диска Windows (или ранее подготовленных загрузочных дискет) и восстановить систему без загрузки компьютера с жесткого диска.

При загрузке системы с дистрибутивного компакт-диска Windows появляется экран, на котором можно выбрать между установкой Windows и восстановлением существующей системы. При выборе второго варианта предлагается вставить дистрибутивный компакт-диск Windows (если он не вставлен в CD-привод). Далее вы должны выбрать один из двух вариантов восстановления: запустить консоль восстановления или начать процесс аварийного восстановления. Если при появлении экрана Setup Welcome (Вас приветствует программа установки) вы нажмете клавишу F10, это меню выводиться не будет, а сразу запустится консоль восстановления.

После запуска консоль восстановления сканирует жесткие диски и формирует список систем Windows NT и Windows на данном компьютере (если они есть). Выбрав нужную систему, вы должны ввести пароль, соответствующий учетной записи администратора для данной системы. Если регистрация прошла успешно, система предоставляет вам командную оболочку, аналогичную среде MS-DOS. Гибкий набор команд позволяет выполнять простые операции с файлами (вроде копирования, переименования и удаления), включать и отключать службы и драйверы и даже восстанавливать MBR и загрузочные записи. Однако консоль восстановления обеспечивает доступ лишь к корневому каталогу, к каталогу, в котором установлена система и в котором вы сейчас зарегистрировались, и к каталогам на сменных дисках, например на компакт-дисках или 3,5-дюймовых дискетах (если это разрешено локальной политикой безопасности, чьи параметры хранятся в кусте SECURITY реестра текущей системы). Эти ограничения диктуются требованиями защиты данных, право на доступ к которым может отсутствовать у администратора одной из систем. Вы можете переопределить эти ограничения, используя редактор локальной политики безопасности (secpol.msc) для настройки параметров Recovery Console (Консоль восстановления) в

папке Security Options (Параметры безопасности) в Local Policies (Локальные политики) при нормальной загрузке системы.

Для поддержки таких команд файлового ввода-вывода, как Cd, Rename или Move, консоль восстановления использует встроенный интерфейс системных вызовов Windows. Команды Enable и Disable, позволяющие изменять режимы запуска драйверов устройств и сервисов (служб), работают иначе. Например, когда вы командуете консоли восстановления отключить какой-либо драйвер, она обращается к разделу реестра Services и присваивает параметру Start в подразделе для соответствующего драйвера значение SERVICE_DISABLED. В результате при следующей загрузке системы данный драйвер загружаться не будет. Консоль также загружает куст реестра SYSTEM (\Windows\System32\Config\System) для текущей системы, где в разделе HKLM\SYSTEM\CurrentControlSet\Services хранится нужная информация.

Когда вы загружаете систему с дистрибутивного компакт-диска Windows или загрузочных дискет, к моменту появления экрана, предлагающего выбор между установкой или восстановлением Windows, происходит загрузка с компакт-диска стартовой копии ядра Windows и всех драйверов поддержки (например, драйверов NTFS, FAT, SCSI и видеоадаптера). На компьютерах с процессорами типа x86 загрузка с компакт-диска управляется файлом Txtsetup.sif из каталога I386. В нем содержится список файлов, подлежащих загрузке, с указанием их местонахождения на компакт-диске. Как и при загрузке Windows с жесткого диска, первой запускаемой программой пользовательского режима является диспетчер сеансов (Smss.exe), расположенный в каталоге I386\System32. Диспетчер сеансов, используемый программой установки Windows, отличается от стандартного в уже установленной системе. Эта версия диспетчера сеансов предоставляет меню, позволяющие установить или восстановить Windows, а также выбрать тип восстановления. В процессе установки Windows этот компонент также помогает выбрать раздел для установки системы и копирует файлы на жесткий диск.

После запуска консоли восстановления диспетчер сеансов загружает и запускает два драйвера устройств, реализующие эту консоль: Spcmdcon.sys и Setupdd.sys. Первый предоставляет интерактивную командную строку и обрабатывает высокоуровневые команды. Второй является драйвером поддержки, предоставляющим Spcmdcon.sys набор функций для управления разделами диска, загрузки кустов реестра и управления видеовыводом. Setupdd.sys также взаимодействует с драйверами дисковых устройств для управления разделами и выводит на экран сообщения, используя базовую видеоподдержку, встроенную в ядро Windows.

Консоль восстановления, приняв от вас пароль для входа в выбранную систему, должна проверить его, даже несмотря на то что подсистема защиты Windows сейчас не функционирует. Таким образом, проверка пароля возлагается исключительно на консоль восстановления. Для этого консоль прежде всего загружает с жесткого диска (через Setupdd.sys) куст реестра диспетчера учетных записей безопасности (Security Accounts Manager, SAM) данной системы, где хранится информация о паролях. Куст SAM находится в ката-

логе \Windows\System32\Config\Sam. После загрузки этого куста консоль восстановления находит в реестре системный ключ для расшифровки копии SAM в памяти. Шифрование куста SAM введено, начиная с Windows NT 4 Service Pack 3, для защиты от попыток взлома из MS-DOS.

Далее консоль восстановления (Spcmdcon.sys) отыскивает в SAM пароль для учетной записи Administrator (Администратор). На заключительном этапе проверки консоль хэширует пароль по алгоритму MD5, а затем сравнивает полученный хэш с зашифрованным хэшем из SAM. Если они совпадают, консоль восстановления считает, что вы успешно вошли в систему, в ином случае консоль восстановления отказывает вам в доступе.

Решение распространенных проблем загрузки

В этом разделе описываются проблемы, которые могут возникнуть в процессе загрузки, их симптомы, причины и подходы к решению.

Повреждение MBR

- **Симптомы** Система с поврежденной главной загрузочной записью (Master Boot Record, MBR) пройдет тест самодиагностики при включении, выполняемый BIOS (power-on self test, POST), выведет на экран информацию о версии BIOS или модели компьютера, затем экран станет черным, и компьютер зависнет. В зависимости от типа повреждения MBR вы можете увидеть одно из следующих сообщений: «Invalid Partition Table» (недопустимая таблица разделов), «Error Loading Operating System» (ошибка при загрузке операционной системы) или «Missing Operating System» (операционная система не найдена).
- **Причина** MBR может быть повреждена из-за ошибок жесткого диска, в результате ошибки одного из драйверов в процессе работы Windows или из-за деструктивной деятельности какого-либо вируса.
- **Решение** Загрузите консоль восстановления и запустите команду *fixmbr*. Эта команда заменяет исполняемый код в MBR. К сожалению, она не исправит таблицу разделов. Единственный способ это сделать — восстановить поврежденную таблицу разделов из резервной копии или использовать сторонний инструмент для устранения повреждений на диске.

Повреждение загрузочного сектора

- **Симптомы** Повреждение загрузочного сектора выглядит как повреждение MBR, когда система зависает с черным экраном после прохождения BIOS POST, либо на черном экране появляется сообщение: «A disk read error occurred» (ошибка чтения с диска), «NTLDR is missing» (NTLDR не найден) или «NTLDR is compressed» (NTLDR заархивирован).
- **Причина** Загрузочная запись может быть повреждена из-за ошибок жесткого диска, в результате ошибки одного из драйверов в процессе работы Windows или из-за деструктивной деятельности какого-либо вируса.

- **Решение** Загрузите консоль восстановления и запустите команду *fixboot*. Эта команда перепишет загрузочный сектор указанного вами тома. Вы должны выполнить такую команду применительно к системному и загрузочному томам, если они разные.

Неправильная конфигурация Boot.ini

- **Симптом** После BIOS POST вы видите сообщение, которое начинается как «Windows could not start because of a computer disk hardware configuration problem» (Windows не удалось запустить из-за проблемы с конфигурацией дискового устройства), «Could not read from selected boot disk» (не удалось считать данные с выбранного загрузочного диска) или «Check boot path and disk hardware» (проверьте путь к загрузочному диску и дисковое устройство).
- **Причина** Файл Boot.ini удален, поврежден или больше не ссылается на загрузочный том из-за добавления раздела, которое привело к изменению ARC-имени тома (Advanced RISC Computing).
- **Решение** Загрузите консоль восстановления и запустите команду *bootcfg /rebuild*. Эта команда заставит консоль восстановления просканировать каждый том в поисках установленных систем Windows. Обнаружив первую из них, она спросит, следует ли добавить ее в Boot.ini как вариант загрузки и под каким названием отображать ее в загрузочном меню.

Повреждение системных файлов

- **Симптомы** Повреждение системных файлов (в том числе драйверов и DLL) может проявляться по-разному. Один из вариантов — сообщение на черном экране после прохождения BIOS POST, в котором говорится «Windows could not start because the following file is missing or corrupt» (Windows не удалось запустить из-за отсутствия или повреждения следующего файла). Далее выводится имя файла и запрос на его переустановку. Еще один вариант — синий экран в результате краха при загрузке с текстом «STOP: 0xC0000135 {Unable to Locate Component}».
- **Причины** Том, на котором находится системный файл, поврежден, один или несколько системных файлов удалены либо повреждены.
- **Решение** Загрузите консоль восстановления и запустите команду *chkdsk*. Эта команда попытается устранить повреждение тома. Если Chkdsk не сообщит о каких-либо проблемах, возьмите резервную копию нужного системного файла. Одно из мест, где можно найти такие копии, — каталог \Windows\System32\DllCache, в котором Windows хранит копии многих системных файлов для использования Windows File Protection (см. врезку «Windows File Protection» далее в этом разделе). Если вам не удалось найти копию файла в этом каталоге, поищите ее на другом компьютере в сети. Заметьте, что резервная копия файла должна быть от того же пакета обновлений или критического исправления, что и заменяемый файл.

В некоторых случаях может быть удалено или повреждено много системных файлов, поэтому процесс восстановления потребует неоднократных перезагрузок, пока вы поочередно не замените все файлы. Если вы считаете, что повреждения системных файлов слишком обширны, подумайте о восстановлении системы с резервного образа, который генерируется, например, Automated System Recovery (ASR). Запустив Windows Backup (Архивация данных) [эта программа находится в папке System (Служебные) в группе Accessories (Стандартные) меню Start (Пуск)], вы можете сгенерировать резервный ASR-образ, который включает все файлы на системном и загрузочном томах, плюс дискету, на которой сохраняется информация о дисках и томах в системе. Чтобы восстановить систему из ASR, загрузите компьютер с дистрибутива Windows и нажмите F2, когда появится соответствующий запрос.

Если у вас нет резервной копии, остается последнее средство — запуск программы установки Windows в режиме исправления: загрузите компьютер с дистрибутива Windows и следуйте указаниям мастера. Мастер спросит вас, хотите ли вы исправить существующую систему или установить новую. Как только вы выберете первый вариант, Setup переустановит все системные файлы, сохранив данные ваших приложений и параметры реестра.

Windows File Protection

Помимо своих основных задач, Winlogon также поддерживает функциональность защиты файлов Windows (Windows File Protection, WFP). WFP, которая реализована в виде двух DLL (\Windows\System32\Sfc.dll и \Windows\System32\Sfc_os.dll), отслеживает несколько каталогов на предмет изменения ключевых драйверов, исполняемых файлов и DLL, в том числе большинство подкаталогов в \Windows. При этом она использует версию *ReadDirectoryChangesW* для Native API. Когда WFP обнаруживает изменение в одном из системных файлов, список которых «зашит» в \Windows\System32\Sfcfiles.dll (с помощью утилиты Strings от *www.sysinternals.com* вы можете получить этот список), она проверяет, подписан ли данный файл цифровой подписью Microsoft (об этом процессе см. раздел «Установка драйверов» главы 9). Если подписан, WFP разрешает изменение и копирует файл в свой резервный каталог. По умолчанию это \Windows\System32\DllCache, но его можно переопределить, изменив параметр реестра HKLM\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\SFCDllCacheDir. Критические исправления и пакеты обновлений всегда устанавливают системные файлы, подписанные Microsoft.

Если в результате модификации появляется файл, не подписанный Microsoft, WFP заменяет его резервной версией из подкаталога DLLCache. Если Winlogon не удается найти резервную версию в этом подкаталоге, он проверяет сетевой путь установки или дистрибутив (в этом случае предлагается вставить компакт-диск).

Повреждение куста System

- **Симптомы** Если куст реестра System (сведения об этом кусте см. в разделе «Реестр» главы 4) отсутствует или поврежден, NTLDR выводит на черном экране после BIOS POST сообщение «Windows could not start because the following file is missing or corrupt: \WINDOWS\SYSTEM32\CONFIG\SYSTEM» (Windows не удалось запустить из-за отсутствия или повреждения следующего файла: \WINDOWS\SYSTEM32\CONFIG\SYSTEM).
- **Причины** Куст реестра System, который содержит конфигурационную информацию, необходимую для загрузки системы, поврежден или удален.
- **Решение** Загрузите консоль восстановления и запустите команду *chkdsk* применительно к загрузочному тому, чтобы исправить любые возможные его повреждения. Если это не решило проблему, возьмите резервную копию куста System. Если вы делали резервные ASR-копии системы или использовали утилиту Windows Backup для создания резервных копий состояния системы (это один из вариантов, предлагаемых в ее UI), то скопируйте кусты реестра из самой последней резервной копии, которые хранятся в каталоге \Windows\Repair, в частности скопируйте файл System в \Windows\System32\Config.

Если вы используете Windows XP и средство System Restore (Восстановление системы) включено (о System Restore см. в главе 12), то можете получить резервные копии кустов реестра, в том числе System, из самой последней точки восстановления. Однако не исключено, что консоль восстановления не позволит вам обратиться к каталогу, где хранятся точки восстановления, — \System Volume Information. Версия консоли восстановления из Windows XP Service Pack 1 разрешает доступ к этому каталогу, а более старые версии — нет (если только это не разрешено локальной политикой безопасности). Чтобы обойти это ограничение, используйте Local Security Policy Editor (Редактор локальной политики безопасности) для изменения параметров консоли восстановления, как уже пояснялось ранее. Вы также можете применить сторонние утилиты для доступа к другим каталогам. Получив доступ к каталогу точек восстановления, выполните следующие операции, чтобы получить файлы кустов реестра.

1. Перейдите в каталоге \System Volume Information на загрузочном томе в подкаталог, имя которого начинается с «_restore».
2. Найдите подкаталог RP с самым большим числом (например, RP173).
3. Скопируйте файл с именем _REGISTRY_MACHINE_SYSTEM в файл \Windows\System32\Config\System.
4. Перезагрузите систему.

Другой вариант — попробовать исправить повреждение с помощью утилиты Microsoft ChkReg. Она пытается делать это автоматически и запускается с дискет, подготовленных в программе установки Windows XP Setup. Эту утилиту и инструкции по ее применению вы найдете по ссылке *http://www.microsoft.com/downloads/details.aspx?displaylang=en&familyid=56d3c201-2c68-4de8-9229-ca494362419c*.

Если у вас нет резервных копий, нет доступа к точкам восстановления и утилита ChkReg не помогла, используйте копию куста System из \Windows \Repair как последнее средство. Windows Setup делает копию куста System по окончании установки, поэтому вы потеряете все изменения в конфигурации системы и драйверов устройств, произошедшие с того момента.

Крах или зависание после вывода экрана-заставки

- **Симптомы** К этой категории относятся проблемы, которые возникают после отображения экрана-заставки Windows, вывода рабочего стола или входа в систему. Они могут проявляться как крах с отображением синего экрана или зависание, при котором замораживается вся система (либо сохраняется возможность перемещать по экрану курсор мыши, но система ни на что не реагирует).
- **Причины** Эти проблемы почти всегда являются следствием ошибки в драйвере устройства, но иногда могут быть результатом повреждения куста реестра, отличного от System.
- **Решение** Вы можете попытаться устранить такую проблему. Первое, что стоит попробовать, — последнюю удачную конфигурацию (last known good, LKG), о которой уже рассказывалось в этой главе и в разделе «Сервисы» главы 4. Она состоит из набора управления реестром (registry control set), с помощью которого в последний раз удалось успешно загрузить систему. Поскольку этот набор включает базовую системную конфигурацию и регистрационную базу данных драйверов устройств и сервисов, он не отражает самые последние изменения в составе драйверов устройств или сервисов, что часто помогает обойти источник проблемы. Для доступа к последней удачной конфигурации нажмите клавишу F8 на самой ранней стадии процесса загрузки и в появившемся загрузочном меню выберите этот вариант.

Как уже говорилось в этой главе, когда вы загружаете LKG, система сохраняет набор управления, от которого вы тем самым отказываетесь, и помечает его как неудачный. Если LKG позволит сделать систему загружаемой, вы сможете экспортировать содержимое текущего и неудачного наборов управления в .reg-файлы и, сравнив их, определить, что стало причиной неудачной загрузки системы. Для этого используйте поддержку экспорта в Regedit, доступную в меню File (Файл) [или Registry (Реестр), если вы работаете с Windows 2000].

1. Запустите Regedit и выберите HKLM\System\CurrentControlSet.
2. Выберите Export (Экспорт) из меню File (Файл) и сохраните содержимое в файл с именем good.reg.
3. Откройте HKLM\System\Select, посмотрите значение параметра Failed и выберите подраздел HKLM\System\ControlXXX, где XXX — значение параметра Failed.
4. Экспортируйте содержимое этого набора управления в файл bad.reg.

5. Используйте Wordpad для глобальной замены всех вхождений «CurrentControlSet» в good.reg на «ControlSet».
6. С помощью Wordpad замените все вхождения «ControlXXX» (вместо XXX должно быть значение параметра Failed) в bad.reg на «ControlSet».
7. Запустите Windiff из Support Tools и сравните два файла.

Различия между неудачным и удачным наборами управления могут быть весьма значительными, поэтому вы должны сосредоточиться на изменениях в подразделе Control, а также в подразделах Parameters каждого драйвера и сервиса, зарегистрированного в подразделе Services. Различия в подразделах Enum разделов для драйверов в ветви Services набора управления игнорируйте.

Если проблема вызвана драйвером или сервисом, который присутствовал в системе до последней успешной загрузки, LKG не сделает систему загружаемой. LKG не поможет и в том случае, если изменившийся проблематичный параметр конфигурации находится вне набора управления или если он был изменен до последней успешной загрузки. В таких случаях следующий шаг — попробовать загрузиться в безопасном режиме (о нем уже рассказывалось в этом разделе). Если система успешно загружается в безопасном режиме и вам известно, какой драйвер привел к провалу нормальной загрузки, вы можете отключить его через диспетчер устройств, доступный с вкладки Hardware (Оборудование) апплета System (Система). Для этого укажите проблемный драйвер и выберите Disable (Отключить) из меню Action (Действие). Если вы используете Windows XP или Windows Server 2003, недавно обновили драйвер и считаете, что в обновленной версии есть какая-то ошибка, то можете выбрать откат драйвера к его предыдущей версии, что также делается в диспетчере устройств. Чтобы восстановить предыдущую версию драйвера, дважды щелкните нужный драйвер для открытия его окна свойств и нажмите кнопку Roll Back Driver (Откатить) на вкладке Drivers (Драйвер).

В Windows XP при включенном System Restore предлагается вариант отката состояния всей системы к предыдущей точке (определенной средством System Restore), когда LKG ничего не дала. Безопасный режим распознает наличие точек восстановления и, если они есть, спрашивает, что вы хотите — войти в систему и вручную выполнить диагностику и исправление или запустить мастер восстановления системы. Попытка сделать систему загружаемой с помощью System Restore — хороший вариант, когда вы знаете причину проблемы и хотите автоматически ее устранить или когда причина вам не известна, но вы не желаете тратить время на ее поиск.

Если System Restore вас не устраивает или если вам нужно определить причину краха при нормальной загрузке, когда в безопасном режиме система успешно загружается, то попробуйте вести журнал в ходе неудачной загрузки. Для этого выберите соответствующий вариант в загрузочном меню, которое открывается нажатием клавиши F8 на самой ранней стадии загрузки. Как уже говорилось в этой главе, диспетчер сеансов (\Windows\System32\Smss.exe) сохраняет журнал загрузки в \Windows\ntbtlog.txt; в нем отмечаются как загруженные, так и незагруженные системой драйверы устройств,

поэтому вы получите такой журнал, только если крах или зависание происходит после инициализации диспетчера сеансов. После перезагрузки в безопасный режим система добавит в существующий журнал загрузки новые записи. Отделите части журнала, относящиеся к неудачной попытке загрузки и к загрузке в безопасный режим, и сохраните их в разных файлах. Удалите строки с текстом «Did not load driver», а затем сравните эти файлы с помощью утилиты наподобие Windiff. Поочередно отключайте драйверы, загружавшиеся при нормальной загрузке, но не в безопасном режиме, пока система вновь не будет загружаться в нормальном режиме. (После чего вновь включите драйверы, не связанные с проблемой.)

Если вам не удается получить журнал при нормальной загрузке (например, из-за того, что крах системы происходит до инициализации диспетчера сеансов), если система рушится и при загрузке в безопасном режиме или если при сравнении двух частей журнала не обнаруживается значимых различий, то остается лишь прибегнуть к утилите Driver Verifier в сочетании с анализом аварийного дампа. (Более подробные сведения по этой тематике см. в главе 14.)

Завершение работы системы

Если в систему кто-то вошел и некий процесс инициирует завершение работы системы, вызывая Windows-функцию *ExitWindowsEx*, Csrss получает сообщение о необходимости завершения системы. Тогда Csrss в интересах инициатора завершения системы посылает скрытому окну, которое принадлежит Winlogon, Windows-сообщение с требованием завершить работу системы. Winlogon, олицетворяющий зарегистрированного в данный момент пользователя (чей контекст защиты может совпадать, а может и не совпадать с контекстом защиты пользователя процесса, инициировавшего завершение работы), вызывает *ExitWindowsEx* с набором специальных внутренних флагов. В результате Csrss получает еще одно сообщение с запросом на завершение системы.

На этот раз Csrss видит, что запрос поступил от Winlogon, и перебирает все процессы в сеансе интерактивного пользователя (а не того, кто инициировал завершение системы). Вызвав *SetProcessShutdownParameters*, процесс может указать уровень завершения (shutdown level), который сообщает системе, когда этому процессу нужно завершиться по отношению к другим процессам. Допустимые уровни укладываются в диапазон 0–1023 (по умолчанию — 640). Explorer, например, устанавливает свой уровень в 2, а Task Manager — в 1. Для каждого процесса, владеющего окном верхнего уровня, Csrss посылает сообщения WM_QUERYENDSESSION всем его потокам с циклом выборки Windows-сообщений. Если поток возвращает TRUE, процесс завершения работы системы продолжается. Тогда Csrss посылает потоку сообщение WM_ENDSESSION с требованием завершить свою работу. Csrss ждет завершения потока в течение времени, указанного в HKCU\Control Panel\ Desktop\HungAppTimeout (по умолчанию — 5000 мс).

Если в течение указанного времени поток не завершается, Csrss открывает диалоговое окно, показанное на рис. 5-5. (Вывод этого окна можно отключить, присвоив параметру HKCU\Control Panel\Desktop\AutoEndTasks значение, равное 1.) Диалоговое окно уведомляет пользователя о том, что корректное завершение данной программы невозможно, и предлагает принудительно завершить процесс или отменить завершение работы системы (таймаут для этого диалогового окна не предусмотрен, а значит, на этом этапе запрос на завершение может ждать бесконечно долго).

Рис. 5-5. *Диалоговое окно для принудительного закрытия программы*

Если поток успевает завершиться до истечения указанного времени, Csrss посылает пары сообщений WM_QUERYENDSESSION/WM_ENDSESSION другим потокам процесса с окнами верхнего уровня. Как только все его потоки завершаются, Csrss завершает выполнение этого процесса и переходит к следующему процессу в интерактивном сеансе.

ЭКСПЕРИМЕНТ: проверка HungAppTimeout

Вы можете проверить, как используется параметр реестра HungAppTimeout, запустив Notepad, введя в него какой-нибудь текст и выйдя из системы. По истечении времени, заданного в HungAppTimeout, Csrss.exe откроет диалоговое окно с запросом о том, хотите ли вы закрыть процесс Notepad, который еще не завершился. Notepad ждет, когда вы сообщите ему, надо ли сохранить введенный вами текст. Если вы нажмете Cancel в этом диалоговом окне, Csrss.exe отменит завершение работы системы.

Обнаружив консольное приложение, Csrss вызывает обработчик консоли, посылая событие CTRL_LOGOFF_EVENT (при завершении работы системы только процессы сервисов получают события CTRL_SHUTDOWN_ EVENT). Если обработчик возвращает FALSE, Csrss уничтожает процесс. Если обработчик возвращает TRUE или не отвечает в течение времени, указанного в HKCU\Control Panel\Desktop\WaitToKillAppTimeout (по умолчанию — 20000 мс), Csrss выводит диалоговое окно, показанное на рис. 5-5.

Далее Winlogon вызывает функцию *ExitWindowsEx*, чтобы Csrss завершил любые COM-процессы, являющиеся частью сеанса интерактивного пользователя.

К этому моменту выполнение всех процессов в сеансе уже завершено. Winlogon снова вызывает функцию *ExitWindowsEx*, на этот раз в контексте

системного процесса, и та посылает Csrss сообщение. Csrss просматривает все процессы, принадлежащие контексту системы и рассылает сообщения WM_QUERYENDSESSION/WM_ENDSESSION всем GUI-потокам. Но консольным приложениям с зарегистрированными обработчиками Csrss посылает не CTRL_LOGOFF_EVENT, а CTRL_SHUTDOWN_EVENT. Заметьте, что SCM является консольной программой, которой регистрируется свой обработчик. Получив запрос на завершение, SCM рассылает соответствующие сообщения всем сервисам, которые зарегистрированы на уведомление о завершении работы. Подробнее о завершении работы сервисов (в том числе о таймауте для SCM) см. раздел «Сервисы» главы 4.

Хотя при завершении системных процессов действуют те же таймауты, что и для пользовательских процессов, Csrss не выводит никаких диалоговых окон и не завершает их принудительно. (Значения таймаутов завершения системных процессов берутся из профиля пользователя по умолчанию.) Смысл этих таймаутов только в том, чтобы системные процессы корректно завершились до выключения системы. Но многие системные процессы вроде Smss, Winlogon, SCM и LSASS на самом деле еще выполняются при выключении системы.

Как только Csrss заканчивает рассылку уведомлений системным процессам о завершении работы, Winlogon вызывает функцию исполнительной системы *NtShutdownSystem*. Она в свою очередь вызывает функцию *NtSetSystemPowerState*, управляющую завершением драйверов и остальных компонентов исполнительной системы (диспетчеров Plug and Play, электропитания, ввода-вывода, конфигурации и памяти).

Например, *NtSetSystemPowerState* вызывает диспетчер ввода-вывода для рассылки пакетов завершения ввода-вывода всем драйверам устройств, запросившим уведомление о завершении системы. Это позволяет драйверам подготовить свои устройства к завершению работы Windows. Диспетчер конфигурации сбрасывает на диск все измененные данные реестра, а диспетчер памяти записывает все измененные страницы с файловыми данными обратно в соответствующие файлы. Диспетчер памяти производит очистку страничного файла (если это указано в настройках). Далее вновь вызывается диспетчер ввода-вывода, который информирует драйверы файловой системы о завершении Windows. Процесс завершения работы заканчивается на диспетчере электропитания, дальнейшие действия которого зависят от пользовательских настроек (выключение компьютера, перезагрузка или переход в ждущий режим).

Резюме

В этой главе мы подробно исследовали процессы загрузки и завершения работы Windows в нормальном режиме и при возникновении сбоев. К этому моменту мы уже рассмотрели общую структуру Windows и базовые системные механизмы, обеспечивающие запуск, работу и в конечном счете выключение системы. Заложив такой фундамент, можно переходить к более подробному изучению отдельных компонентов исполнительной системы, начиная с процессов и потоков.

ГЛАВА 6

Процессы, потоки и задания

В этой главе мы рассмотрим структуры данных и алгоритмы, связанные с процессами, потоками и заданиями в Microsoft Windows. В первом разделе основное внимание уделяется внутренним структурам данных, из которых состоит процесс. Во втором разделе поясняются этапы создания процесса (и его первичного потока). Далее поясняются внутреннее устройство потоков и их планирование. Завершается глава описанием объекта «задание» (job object).

В процессе изложения материала мы будем упоминать соответствующие счетчики производительности или переменные ядра. Хотя программирование под Windows не является предметом этой книги, в ней перечисляются Windows-функции, связанные с процессами, потоками и заданиями, — это даст вам представление о том, как они используются.

Внутреннее устройство процессов

В этом разделе описываются ключевые структуры данных процессов Windows. Также поясняются основные переменные ядра, счетчики производительности, функции и утилиты, имеющие отношение к процессам.

Структуры данных

Каждый процесс в Windows представлен блоком процесса, создаваемым исполнительной системой (EPROCESS). Кроме многочисленных атрибутов, относящихся к процессу, в блоке EPROCESS содержатся указатели на некоторые структуры данных. Так, у каждого процесса есть один или более потоков, представляемых блоками потоков исполнительной системы (ETHREAD) (см. раздел «Внутреннее устройство потоков» далее в этой главе). Блок EPROCESS и связанные с ним структуры данных — за исключением блока переменных окружения процесса (process environment block, PEB) — существуют в системном пространстве. PEB находится в адресном пространстве процесса, так как содержит данные, модифицируемые кодом пользовательского режима.

Для каждого процесса, выполняющего Windows-программу, процесс подсистемы Windows (Csrss) поддерживает в дополнение к блоку EPROCESS параллельную структуру данных. Кроме того, часть подсистемы Windows, работающая в режиме ядра (Win32k.sys), поддерживает структуру данных для каждого процесса, которая создается при первом вызове потоком любой функции USER или GDI, реализованной в режиме ядра.

На рис. 6-1 показана упрощенная схема структур данных процесса и потока. Каждая из этих структур детально рассматривается далее в этой главе.

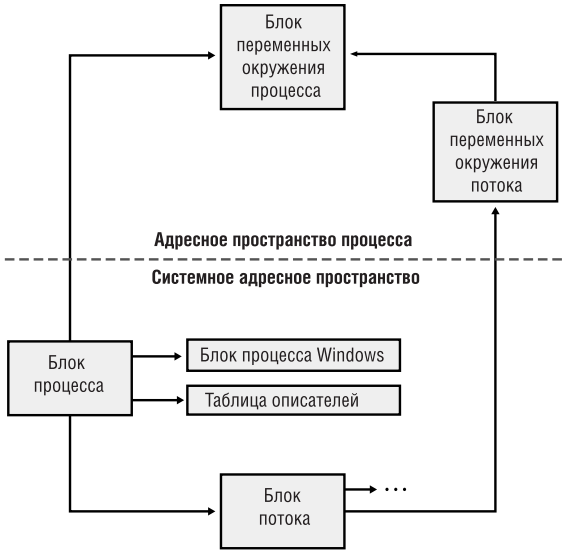


Рис. 6-1. *Структуры данных, сопоставляемые с процессами и потоками*

Сначала рассмотрим блок процесса. (Изучение блока потока мы отложим до раздела «Внутреннее устройство потоков».) Ключевые поля EPROCESS показаны на рис. 6-2.

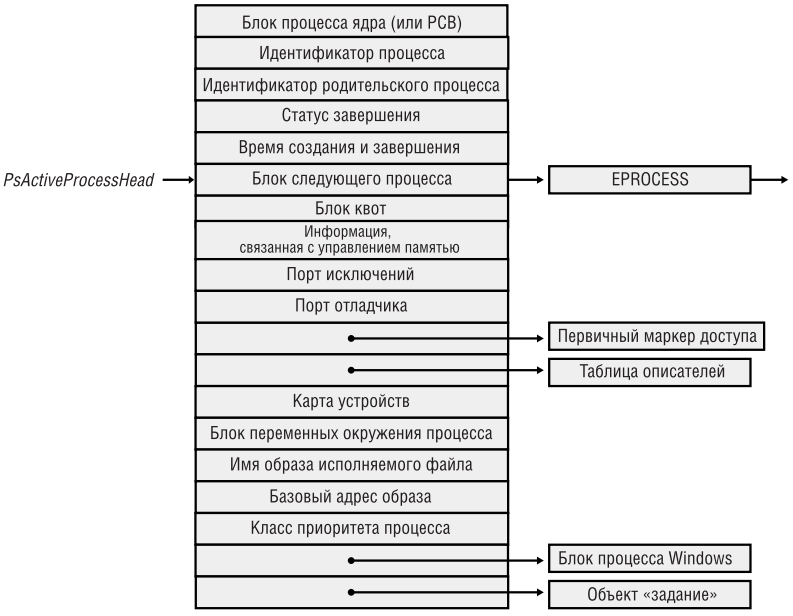


Рис. 6-2. *Блок процесса (EPROCESS), создаваемый исполнительной системой*

ЭКСПЕРИМЕНТ: исследуем формат блока EPROCESS

Список полей, составляющих блок EPROCESS, и их смещения в шестнадцатеричной форме, можно увидеть с помощью команды *dt eprocess* отладчика ядра (подробнее об отладчике ядра см. главу 1). Вот что дает эта команда (вывод обрезан для экономии места):

```
lkd> dt _eprocess
nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x06c ProcessLock      : _EX_PUSH_LOCK
   +0x070 CreateTime       : _LARGE_INTEGER
   +0x078 ExitTime         : _LARGE_INTEGER
   +0x080 RundownProtect   : _EX_RUNDOWN_REF
   +0x084 UniqueProcessId  : Ptr32 Void
   +0x088 ActiveProcessLinks : _LIST_ENTRY
   +0x090 QuotaUsage       : [3] Uint4B
   +0x09c QuotaPeak        : [3] Uint4B
   +0x0a8 CommitCharge     : Uint4B
   +0x0ac PeakVirtualSize  : Uint4B
   +0x0b0 VirtualSize      : Uint4B
   +0x0b4 SessionProcessLinks : _LIST_ENTRY
   +0x0bc DebugPort        : Ptr32 Void
   +0x0c0 ExceptionPort    : Ptr32 Void
   +0x0c4 ObjectTable      : Ptr32 _HANDLE_TABLE
   +0x0c8 Token            : _EX_FAST_REF
   +0x0cc WorkingSetLock   : _FAST_MUTEX
   +0x0ec WorkingSetPage   : Uint4B
   +0x0f0 AddressCreationLock : _FAST_MUTEX
   +0x110 HyperSpaceLock   : Uint4B
   +0x114 ForkInProgress   : Ptr32 _ETHREAD
   +0x118 HardwareTrigger  : Uint4B
```

Заметьте, что первое поле (Pcb) на самом деле является подструктурой — блоком процесса, принадлежащим ядру (KPROCESS). Именно здесь хранится информация, используемая при планировании. Для вывода формата блока процесса KPROCESS введите **dt_kprocess**:

```
lkd> dt _kprocess
nt!_KPROCESS
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 ProfileListHead  : _LIST_ENTRY
   +0x018 DirectoryTableBase : [2] Uint4B
   +0x020 LdtDescriptor    : _KGDTENTRY
   +0x028 Int21Descriptor  : _KIDTENTRY
   +0x030 IopmOffset       : Uint2B
   +0x032 Iopl             : UChar
   +0x033 Unused           : UChar
   +0x034 ActiveProcessors : Uint4B
```

см. след. стр.

```
+0x038 KernelTime       : Uint4B
+0x03c UserTime         : Uint4B
+0x040 ReadyListHead    : _LIST_ENTRY
+0x048 SwapListEntry    : _SINGLE_LIST_ENTRY
+0x04c VdmTrapcHandler  : Ptr32 Void
+0x050 ThreadListHead   : _LIST_ENTRY
+0x058 ProcessLock      : Uint4B
+0x05c Affinity         : Uint4B
+0x060 StackCount       : Uint2B
+0x062 BasePriority     : Char
+0x063 ThreadQuantum    : Char
+0x064 AutoAlignment    : UChar
+0x065 State            : UChar
+0x066 ThreadSeed       : UChar
+0x067 DisableBoost     : UChar
+0x068 PowerState       : UChar
+0x069 DisableQuantum   : UChar
+0x06a IdealNode        : UChar
+0x06b Spare            : UChar
```

Другой способ просмотра KPROCESS (и прочих подструктур в EPROCESS) — использовать ключ рекурсии (*–r*) в команде *dt*. Например, введя **dt _eprocess –r1**, вы увидите все подструктуры с глубиной вложения, равной 1.

Команда *dt* показывает формат блока процесса, но не его содержимое. Чтобы вывести экземпляр самого процесса, можно указать адрес структуры EPROCESS в качестве аргумента команды *dt*. Команда *!process 0 0* позволяет получить адрес всех блоков EPROCESS в системе. Пример вывода этой команды будет приведен далее в этой главе.

Некоторые поля, показанные в предыдущем эксперименте, поясняются в таблице 6-1. Процессы и потоки — неотъемлемая часть Windows, о которой нельзя рассказать, не упомянув множество других компонентов системы. Но, чтобы эта глава не слишком разбухла, мы поясняем механизмы, связанные с процессами и потоками (вроде управления памятью, защиты, объектов и описателей), в других главах.

Таблица 6-1. *Содержимое блока EPROCESS*

Элемент	Описание	Дополнительные ссылки
Блок процесса ядра (KPROCESS)	Общий заголовок объекта диспетчера, указатель на каталог страниц процесса, список блоков потоков ядра (KTHREAD), принадлежащих процессу, базовый приоритет по умолчанию, выделяемый квант процессорного времени, маска привязки к процессорам, а также суммарное время работы потоков в режиме ядра и пользовательском режиме	Раздел «Планирование потоков» далее в этой главе

Таблица 6-1. (*продолжение*)

Элемент	Описание	Дополнительные ссылки
Идентификация процесса	Уникальный идентификатор процесса, идентификатор процесса-создателя, имя выполняемого образа, имя WindowStation-объекта для данного процесса	
Блок квот	Лимиты на пулы подкачиваемой и неподкачиваемой памяти, а также на использование страничного файла плюс текущее и пиковое использование пулов подкачиваемой и неподкачиваемой памяти процесса. Некоторые процессы могут разделять эту структуру: все системные процессы ссылаются на один общесистемный блок квот по умолчанию, а все процессы интерактивного сеанса используют один блок квот, определяемый Winlogon	
Дескрипторы виртуальных адресов (virtual address descriptors, VAD)	Наборы структур данных, описывающих статус частей адресного пространства, которые существуют в процессе	Глава 7
Информация о рабочем наборе	Указатель на список рабочего набора (структура MMWSL); текущий, пиковый, минимальный и максимальный размеры рабочего набора; время последнего усечения (last trim time); счетчик числа ошибок страниц; приоритет памяти; некоторые специфические флаги; хронология ошибок страниц	Глава 7
Информация о виртуальной памяти	Текущий и пиковый размер виртуальной памяти, использование страничного файла, элемент аппаратной таблицы страниц, указывающий на каталог страниц процесса	Глава 7
LPC-порт исключений	Канал межпроцессной связи, по которому диспетчер процессов посылает сообщение, если один из потоков этого процесса вызывает исключение	Раздел «Диспетчеризация исключений» главы 3
Отладочный LPC-порт	Канал межпроцессной связи, по которому диспетчер процессов посылает сообщение, если один из потоков этого процесса генерирует событие отладки	Раздел «LPC» главы 3
Маркер доступа (ACCESS_TOKEN)	Объект исполнительной системы, описывающий профиль защиты данного процесса	Глава 8
Таблица описателей	Адрес таблицы описателей, принадлежащей процессу	Раздел «Описатели объектов и таблица описателей, принадлежащая процессу» главы 3

см. след. стр.

Таблица 6-1. (*окончание*)

Элемент	Описание	Дополнительные ссылки
Карта устройств	Адрес каталога объектов для разрешения ссылок на устройства по именам (поддерживается несколько пользователей)	Раздел «Диспетчер объектов» главы 3
Блок переменных окружения процесса (PEB)	Информация об образе исполняемого файла (базовый адрес, номера версий, список модулей), информация о куче процесса и TLS-памяти (указатели на кучи процесса начинаются с первого байта после PEB)	Далее в этой главе
Блок процесса подсистемы Windows (W32PROCESS)	Дополнительная информация о процессе, необходимая той части подсистемы Windows, которая работает в режиме ядра	

Блок KPROCESS, входящий в блок EPROCESS, и PEB, на который указывает EPROCESS, содержат дополнительные сведения об объекте «процесс». Блок KPROCESS, иногда называемый блоком управления процессом (process control block, PCB), показан на рис. 6-3. Он содержит базовую информацию, нужную ядру Windows для планирования потоков. (О каталогах страниц см. главу 7.)

PEB, размещаемый в адресном пространстве пользовательского процесса, содержит информацию, необходимую загрузчику образов, диспетчеру кучи и другим системным DLL-модулям Windows для доступа из пользовательского режима. (Блоки EPROCESS и KPROCESS доступны только из режима ядра.) Базовая структура PEB, показанная на рис. 6-4, подробнее объясняется далее в этой главе.

Рис. 6-3. *Блок процесса исполнительной системы*

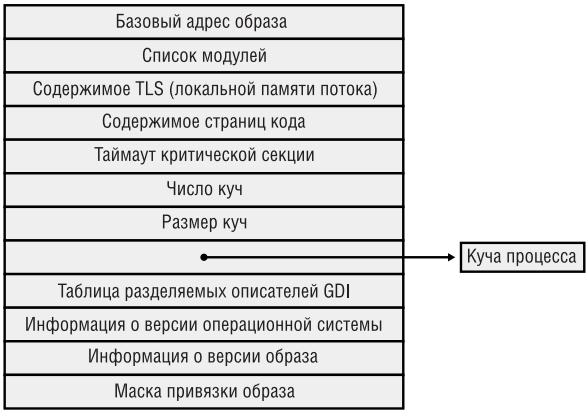


Рис. 6-4. *Поля в блоке PEB*

ЭКСПЕРИМЕНТ: исследуем PEB

Дамп структуры PEB можно получить с помощью команды *!peb* отладчика ядра. Чтобы узнать адрес PEB, используйте команду *!process* так:

```
lkd> !process
PROCESS 8575f030  SessionId: 0  Cid: 08d0    Peb: 7ffdf000
   ParentCid: 0360 DirBase: 1a81b000  ObjectTable: e12bd418
   HandleCount:  66.
   Image: windbg.exe
```

Затем укажите этот адрес в команде *!peb*:

```
lkd> !peb 7ffdf000
PEB at 7ffdf000
   InheritedAddressSpace:    No
   ReadImageFileExecOptions: No
   BeingDebugged:            No
   ImageBaseAddress:         01000000
   Ldr                       00181e90
   Ldr.Initialized:          Yes
   Ldr.InInitializationOrderModuleList: 00181f28 . 00183188
   Ldr.InLoadOrderModuleList:           00181ec0 . 00183178
   Ldr.InMemoryOrderModuleList:         00181ec8 . 00183180
            Base TimeStamp                     Module
       1000000 40478dbd Mar 04 15:12:45 2004 C:\Program Files\Debugging
       Tools for
             Windows\windbg.exe
        77f50000 3eb1b41a May 01 19:56:10 2003 C:\WINDOWS\System32\ntdll.dll
        77e60000 3d6dfa28 Aug 29 06:40:40 2002 C:\WINDOWS\system32\kernel32.dll
        2000000 40476db2 Mar 04 12:56:02 2004 C:\Program Files\Debugging
        Tools for
             Windows\dbgeng.dll        .
```

см. след. стр.

```
    SubSystemData:     00000000
    ProcessHeap:       00080000
    ProcessParameters: 00020000
    WindowTitle:  'C:\Documents and Settings\All Users\Start
    Menu\Programs\Debugging
         Tools for Windows\WinDbg.lnk'
    ImageFile:    'C:\Program Files\Debugging Tools for Windows\windbg.exe'
    CommandLine:  '"C:\Program Files\Debugging Tools for Windows\windbg.exe" '
    DllPath:      'C:\Program Files\Debugging Tools for
Windows;C:\WINDOWS\System32;C:\WINDOWS\system;C:\WINDOWS;.;C:\Program
Files\Windows Resource
Kits\Tools\;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\Program
Files\Support Tools\;c:\sysint;C:\Program Files\ATI Technologies\ATI
Control Panel;C:\Program Files\Resource
Kit\;C:\PROGRA~1\CA\Common\SCANEN~1;C:\PROGRA~1\CA\eTrust\ANTIVI~1;C:\Program
Files\Common Files\Roxio Shared\DLLShared;C:\SFU\common\'
    Environment:  00010000
        =::=::\
        ALLUSERSPROFILE=C:\Documents and Settings\All Users
        APPDATA=C:\Documents and Settings\dsolomon\Application Data
    ...
```

Переменные ядра

В таблице 6-2 перечислено несколько важнейших глобальных переменных ядра, связанных с процессами. На эти переменные мы будем ссылаться по ходу изложения материала, в частности при описании этапов создания процесса.

Таблица 6-2. *Переменные ядра, связанные с процессами*

Переменная	Тип	Описание
PsActiveProcessHead	Заголовок очереди	Заголовок списка блоков процесса
PsIdleProcess	EPROCESS	Блок процесса Idle
PsInitialSystemProcess	Указатель на EPROCESS	Указатель на блок начального системного процесса (с идентификатором 2), который содержит системные потоки
PspCreateProcessNotifyRoutine	Массив указателей	Указатели на процедуры (максимум 8), вызываемые при создании и удалении процесса
PspCreateProcessNotifyRoutineCount	DWORD	Счетчик зарегистрированных процедур уведомления о создании процесса
PspLoadImageNotifyRoutine	Массив указателей	Указатели на процедуры, вызываемые при загрузке образа исполняемого файла
PspLoadImageNotifyRoutineCount	DWORD	Счетчик зарегистрированных процедур уведомления о загрузке образа
PspCidTable	Указатель на HANDLE_TABLE	Таблица описателей для клиентских идентификаторов процесса и потока

Счетчики производительности

Windows поддерживает несколько счетчиков, которые позволяют отслеживать процессы, выполняемые в системе; данные этих счетчиков можно считывать программно или просматривать с помощью оснастки Performance. В таблице 6-3 перечислены счетчики производительности, имеющие отношение к процессам (кроме счетчиков, связанных с управлением памятью и вводом-выводом, которые описываются в главах 7 и 9 соответственно).

Таблица 6-3. *Счетчики производительности, связанные с процессами*

Объект: счетчик	Описание
Process: % Privileged Time (Процесс: % работы в привилегированном режиме)	Процентная доля времени, в течение которого потоки данного процесса выполнялись в режиме ядра
Process: % Processor Time (Процесс: % загруженности процессора)	Процентная доля процессорного времени, использованная потоками процесса за определенный период времени; вычисляется как сумма % Privileged Time и % User Time
Process: % User Time (Процесс: % работы в пользовательском режиме)	Процентная доля времени, в течение которого потоки данного процесса выполнялись в пользовательском режиме
Process: Elapsed Time (Процесс: Прошло времени)	Суммарное время (в секундах), прошедшее с момента создания процесса
Process: ID Process (Процесс: Идентификатор процесса)	Идентификатор процесса; полученное таким образом значение действительно лишь на время выполнения процесса, поскольку идентификаторы могут использоваться повторно
Process: Creating Process ID [Процесс: Код (ID) создавшего процесса]	Идентификатор родительского процесса; его значение не обновляется после завершения родительского процесса
Process: Thread Count (Процесс: Счетчик потоков)	Число потоков в процессе
Process: Handle Count (Процесс: Счетчик дескрипторов)	Число открытых процессом описателей

Сопутствующие функции

В таблице 6-4 приведена информация по некоторым Windows-функциям, связанным с процессами. Более подробные сведения см. в документации Windows API в MSDN Library.

Таблица 6-4. *Функции, связанные с процессами*

Функция	Описание
CreateProcess	Создает новый процесс и поток с использованием идентификации защиты вызывающего процесса
CreateProcessAsUser	Создает новый процесс и поток с указанным альтернативным маркером защиты

см. след. стр.

Таблица 6-4. (*окончание*)

Функция	Описание
CreateProcessWithLogonW	Создает новый процесс и поток для выполнения под учетной записью, соответствующей указанным имени и паролю пользователя
CreateProcessWithTokenW	Создает новый процесс и поток с указанным альтернативным маркером защиты и поддерживает дополнительные возможности, например разрешает загрузку профиля пользователя
OpenProcess	Возвращает описатель указанного объекта «процесс»
ExitProcess	Завершает процесс с уведомлением всех подключенных DLL
TerminateProcess	Завершает процесс без уведомления подключенных DLL
FlushInstructionCache	Опустошает кэш команд указанного процесса
GetProcessTimes	Получает временные параметры процесса, описывающие, сколько времени процесс провел в режиме ядра и в пользовательском режиме
GetExitCodeProcess	Возвращает код завершения процесса, указывающий, как и почему завершился этот процесс
GetCommandLine	Возвращает указатель на командную строку, переданную текущему процессу
GetCurrentProcess	Возвращает псевдоописатель текущего процесса
GetCurrentProcessId	Возвращает идентификатор текущего процесса
GetProcessVersion	Возвращает старший и младший номера версии Windows, необходимой для запуска указанного процесса
GetStartupInfo	Возвращает содержимое структуры STARTUPINFO, заданное при вызове *CreateProcess*
GetEnvironmentStrings	Возвращает адрес блока переменных окружения
GetEnvironmentVariable	Возвращает значение указанной переменной окружения
GetProcessShutdownParameters и *SetProcessShutdownParameters*	Определяет приоритет завершения и число попыток для текущего процесса
GetGuiResources	Возвращает число открытых описателей USER и GDI

ЭКСПЕРИМЕНТ: применение команды *!process* отладчика ядра

Эта команда выводит подмножество информации из блока EPROCESS. Ее вывод для каждого процесса делится на две части. Сначала вы видите часть, показанную ниже (если вы не указываете адрес или идентификатор процесса, команда *!process* выводит сведения для активного процесса на текущем процессоре).

```
lkd> !process
PROCESS 8575f030  SessionId: 0  Cid: 08d0    Peb: 7ffdf000
    ParentCid: 0360
```

```
DirBase: 1a81b000  ObjectTable: e12bd418  HandleCount:  65.
Image: windbg.exe
VadRoot 857f05e0 Vads 71 Clone 0 Private 1152. Modified 98.
Locked 1.
DeviceMap e1e96c88
Token                             e1f5b8a8
ElapsedTime                       1:23:06.0219
UserTime                          0:00:11.0897
KernelTime                        0:00:07.0450
QuotaPoolUsage[PagedPool]         38068
QuotaPoolUsage[NonPagedPool]      2840
Working Set Sizes (now,min,max)  (2552, 50, 345) (10208KB, 200KB, 1380KB)
PeakWorkingSetSize                2715
VirtualSize                       41 Mb
PeakVirtualSize                   41 Mb
PageFaultCount                    3658
MemoryPriority                    BACKGROUND
BasePriority                      8
CommitCharge                      1566
```

Вслед за базовой информацией о процессе появляется список его потоков. Данная часть поясняется в эксперименте «Применение команды *!thread* отладчика ядра» далее в этой главе. Еще одна команда, позволяющая получить информацию о процессе, — *!handle*. Она создает дамп таблицы описателей, принадлежащей процессу (см. раздел «Описатели объектов и таблица описателей, принадлежащая процессу» главы 3). Структуры защиты процессов и потоков описываются в главе 8.

Что делает функция *CreateProcess*

К этому моменту мы уже рассмотрели структуры, которые являются частью процесса, и API-функции, позволяющие вам (и операционной системе) манипулировать процессами. Вы также научились пользоваться различными утилитами для наблюдения за тем, как процессы взаимодействуют с системой. Но как эти процессы появляются на свет и как они завершаются, выполнив задачи, для которых они предназначались? В следующих разделах вы узнаете, как порождаются Windows-процессы.

Создание Windows-процесса осуществляется вызовом одной из таких функций, как *CreateProcess*, *CreateProcessAsUser*, *CreateProcessWithTokenW* или *CreateProcessWithLogonW*, и проходит в несколько этапов с участием трех компонентов операционной системы: Kernel32.dll (библиотеки клиентской части Windows), исполнительной системы и процесса подсистемы окружения Windows (Csrss). Поскольку архитектура Windows поддерживает несколько подсистем окружения, операции, необходимые для создания объекта «процесс» исполнительной системы (которым могут пользоваться и дру-

гие подсистемы окружения), отделены от операций, требуемых для создания Windows-процесса. Поэтому часть действий Windows-функции *CreateProcess* специфична для семантики, привносимой подсистемой Windows.

В приведенном ниже списке перечислены основные этапы создания процесса Windows-функцией *CreateProcess*. Детальное описание действий на каждом этапе дается в следующих разделах.

> **ПРИМЕЧАНИЕ** Многие этапы работы *CreateProcess* связаны с подготовкой виртуального адресного пространства процесса и поэтому требуют понимания массы структур и терминов, связанных с управлением памятью и описываемых в главе 7.

1. Открывается файл образа (EXE), который будет выполняться в процессе.
2. Создается объект «процесс» исполнительной системы.
3. Создается первичный поток (стек, контекст и объект «поток» исполнительной системы).
4. Подсистема Windows уведомляется о создании нового процесса и потока.
5. Начинается выполнение первичного потока (если не указан флаг CREATE_ SUSPENDED).
6. В контексте нового процесса и потока инициализируется адресное пространство (например, загружаются требуемые DLL) и начинается выполнение программы.

Общая схема создания процесса в Windows показана на рис. 6-5.

Прежде чем открыть исполняемый образ для выполнения, CreateProcess делает следующее.

- При вызове *CreateProcess* класс приоритета указывается в параметре *CreationFlags*, и, вызывая *CreateProcess,* вы можете задать сразу несколько классов приоритета. Windows выбирает самый низкий из них.
- Когда для нового процесса не указывается класс приоритета, по умолчанию принимается Normal, если только класс приоритета процесса-создателя не равен Idle или Below Normal. В последнем случае новый процесс получает тот же класс приоритета, что и у родительского процесса.
- Если для нового процесса указан класс приоритета Real-time, а создатель не имеет привилегии Increase Scheduling Priority, устанавливается класс приоритета High. Иначе говоря, функция *CreateProcess* завершается успешно, даже если у того, кто ее вызвал, недостаточно привилегий для создания процессов с классом приоритета Real-time, — просто класс приоритета нового процесса будет ниже Real-time.
- Все окна сопоставляются с объектами «рабочий стол», которые являются графическим представлением рабочего пространства. Если при вызове *CreateProcess* не указан конкретный объект «рабочий стол», новый процесс сопоставляется с текущим объектом «рабочий стол» процесса-создателя.

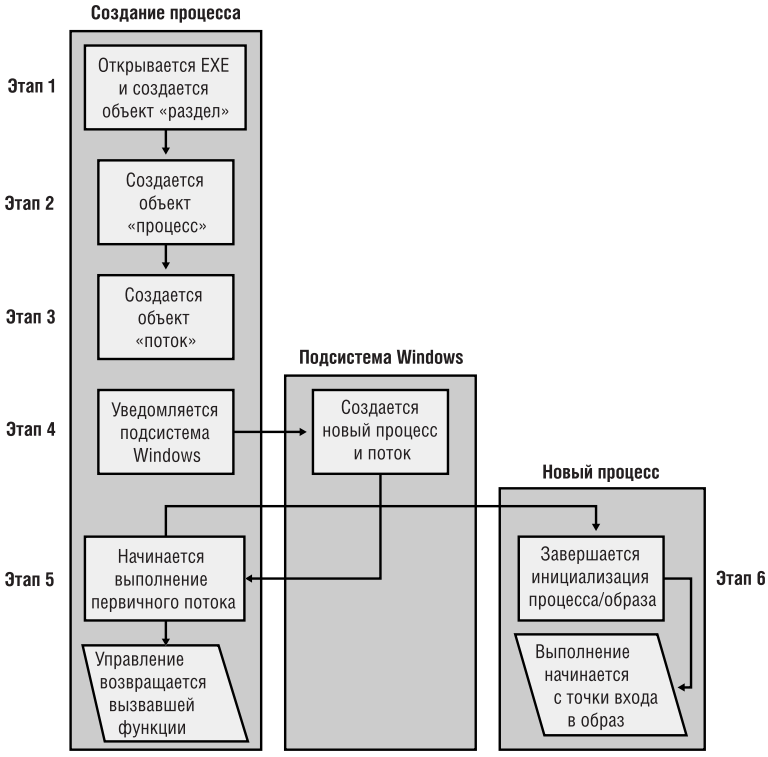


Рис. 6-5. *Основные этапы создания процесса*

Этап 1: открытие образа, подлежащего выполнению

Как показано на рис. 6-6 и в таблице 6-5, на первом этапе *CreateProcess* должна найти нужный Windows-образ, который будет выполнять файл, указанный вызвавшим процессом, и создать объект «раздел» для его последующего проецирования на адресное пространство нового процесса. Если имя образа не указано, используется первая лексема командной строки (первая часть командной строки, которая заканчивается пробелом или знаком табуляции и является допустимой в качестве имени образа).

В Windows XP и Windows Server 2003 *CreateProcess* проверяет, не запрещает ли политика безопасности на данной машине запуск этого образа (см. главу 8).

Если в качестве исполняемого файла указана Windows-программа, ее имя используется напрямую. А если исполняемый файл является не Windows-приложением, а программой MS-DOS, Win16 или POSIX, то *CreateProcess* ищет образ поддержки (support image) для запуска этой программы. Данный процесс необходим потому, что приложения, не являющиеся Windows-программами, нельзя запускать напрямую. Вместо этого Windows использует один из нескольких специальных образов поддержки, которые и отвечают за запуск

приложений, отличных от Windows-программ. Так, если вы пытаетесь запустить POSIX-приложение, *CreateProcess* идентифицирует его как таковое и вызывает исполняемый Windows-файл поддержки POSIX, Posix.exe. А если вы запускаете программу MS-DOS или Win16, стартует исполняемый Windows-файл поддержки Ntvdm.exe. Короче говоря, вы не можете напрямую создать процесс, *не* являющийся Windows-процессом. Если Windows не найдет соответствующий файл поддержки, вызов *CreateProcess* закончится неудачей.

Рис. 6-6. *Выбор активизируемого Windows-образа*

Таблица 6-5. *Дерево решений CreateProcess на первом этапе*

Если файл относится к	Запускается образ поддержки	И происходит
POSIX	Posix.exe	Возврат к первому этапу *CreateProcess*
MS-DOS (в случае файла с расширением EXE, COM или PIF)	Ntvdm.exe	Возврат к первому этапу *CreateProcess*
Win16	Ntvdm.exe	Возврат к первому этапу *CreateProcess*
К командной оболочке (в случае файла с расширением BAT или CMD)	Cmd.exe	Возврат к первому этапу *CreateProcess*

Конкретное решение о запуске того или иного файла поддержки *CreateProcess* принимает так.

- Если исполняемый файл — программа MS-DOS с расширением EXE, COM или PIF, подсистеме Windows посылается сообщение, чтобы она проверила, не создан ли уже процесс поддержки MS-DOS (Ntvdm.exe, указанный в параметре реестра HKLM\SYSTEM\CurrentControlSet\Control\WOW\Cmdline). Если да, этот процесс используется для запуска программы MS-DOS — подсистема Windows посылает виртуальной DOS-машине (Virtual

DOS Machine, VDM) сообщение для запуска новой программы, — после чего управление возвращается к *CreateProcess*. Если нет, запускается Ntvdm.exe и повторно выполняется первый этап *CreateProcess*.

- Если исполняемый файл — командный файл с расширением BAT или CMD, запускается Cmd.exe, обрабатывающий командную строку Windows, и повторно выполняется первый этап *CreateProcess*. (Имя командного файла передается Cmd.exe как первый параметр.)
- Если исполняемый файл — приложение Win16 (Windows 3.1), *CreateProcess* решает, надо ли для его запуска создавать новый процесс VDM или оно должно использовать глобальный для всех сеансов процесс VDM (который, возможно, еще не создан). Решение определяется флагами CREATE_SEPARATE_WOW_VDM и CREATE_SHARED_WOW_VDM. Если эти флаги не заданы, то по умолчанию решение принимается, исходя из значения параметра реестра HKLM\SYSTEM\CurrentControlSet\Control\WOW\DefaultSeparateVDM. Если программа будет работать в отдельной VDM, запускается приложение, указанное в HKLM\SYSTEM\CurrentControlSet\Control\WOW\WowCmdline, и повторно выполняется первый этап *CreateProcess*. В ином случае подсистема Windows посылает сообщение для проверки возможности использования общего процесса VDM. (Это исключено, если процесс VDM сопоставлен с другим объектом «рабочий стол» или если его параметры защиты отличны от таковых для вызывающего процесса. Тогда нужно создавать новый процесс VDM.) Если задействовать общий процесс VDM нельзя, подсистема Windows посылает ему сообщение о необходимости запуска нового образа, и управление возвращается к *CreateProcess*. Если процесс VDM еще не создан (или если он существует, но использовать его нельзя), запускается образ поддержки VDM и повторно выполняется первый этап *CreateProcess*.

К этому моменту *CreateProcess* успешно открывает допустимый исполняемый файл Windows и создает для него объект «раздел». Этот объект еще не спроецирован на память, но уже открыт. Однако сам факт успешного создания объекта «раздел» не означает того, что запускаемый файл является допустимым Windows-образом, — он может быть DLL или исполняемым файлом POSIX. Если это исполняемый файл POSIX, запускается Posix.exe, и *CreateProcess* заново выполняет действия первого этапа. А если это DLL, вызов *CreateProcess* заканчивается неудачей.

CreateProcess, найдя допустимый исполняемый Windows-образ, ищет в разделе реестра HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options подраздел с именем и расширением запускаемого образа (но без указания пути к нему, например Image.exe). Если такой подраздел есть, *CreateProcess* ищет в нем параметр Debugger. Если он присутствует, его значение становится именем запускаемого образа, после чего следует повторение первого этапа *CreateProcess*.

СОВЕТ Вы можете извлечь выгоду из такого поведения *CreateProcess*. Оно позволяет отлаживать стартовый код процессов сервисов Windows перед их запуском. Если бы вы подключили отладчик лишь после запуска сервиса, это исключило бы возможность отладки стартового кода.

Этап 2: создание объекта «процесс»

К началу второго этапа функция *CreateProcess* уже открыла допустимый исполняемый файл Windows и создала объект «раздел» для его проецирования на адресное пространство нового процесса. После этого она создает объект «процесс», чтобы запустить образ вызовом внутренней функции *NtCreateProcess*. Создание объекта «процесс» исполнительной системы включает следующие подэтапы:

- формируется блок EPROCESS;
- создается начальное адресное пространство процесса;
- инициализируется блок процесса ядра (KPROCESS);
- инициализируется адресное пространство процесса (в том числе список рабочего набора и дескрипторы виртуального адресного пространства), а также проецируется образ на это пространство;
- формируется блок PEB;
- завершается инициализация объекта «процесс» исполнительной системы.

ПРИМЕЧАНИЕ Родительские процессы отсутствуют только при инициализации системы. Далее они всегда используются для задания контекстов защиты новых процессов.

Этап 2A: формирование блока EPROCESS

Этот подэтап включает девять операций.

1. Создается и инициализируется блок EPROCESS.
2. От родительского процесса наследуется маска привязки к процессорам.
3. Минимальный и максимальный размеры рабочего набора процесса устанавливаются равными значениям переменных *PsMinimumWorkingSet* и *PsMaximumWorkingSet*.
4. Блок квот нового процесса настраивается на адрес блока квот его родительского процесса и увеличивается счетчик ссылок на блок квот последнего.
5. Наследуется пространство имен устройств Windows (в том числе определение букв дисков, COM-портов и т. д.).
6. В поле *InheritedFromUniqueProcessId* нового объекта «процесс» сохраняется идентификатор родительского процесса.
7. Создается основной маркер доступа процесса (копированием аналогичного маркера родительского процесса). Новый процесс наследует про-

филь защиты своих родителей. Если используется функция *CreateProcessAsUser*, чтобы задать для нового процесса другой маркер доступа, он соответственно модифицируется.

8. Инициализируется таблица описателей, принадлежащая процессу. Если установлен флаг наследования описателей родительского процесса, наследуемые описатели из его таблицы копируются в новый процесс (о таблицах описателей см. главу 3).
9. Статус завершения нового процесса устанавливается как STATUS_PENDING.

Этап 2B: создание начального адресного пространства процесса

Начальное адресное пространство процесса состоит из следующих страниц:

- каталога страниц (этих каталогов может быть больше одного в системах, где таблицы страниц имеют более двух уровней, например в x86-системах в режиме PAE или в 64-разрядных системах);
- страницы гиперпространства;
- списка рабочего набора.

Для создания этих страниц выполняются операции, перечисленные ниже.

1. В соответствующих таблицах страниц формируются записи, позволяющие проецировать эти начальные страницы. Количество страниц вычитается из переменной ядра *MmTotalCommittedPages* и добавляется к переменной ядра *MmProcessCommit*.
2. Из *MmResidentAvailablePages* вычитается минимальный размер рабочего набора по умолчанию (*PsMinimumWorkingSet*).
3. На адресное пространство процесса проецируются страницы таблицы страниц для неподкачиваемой части системного пространства и системного кэша.

Этап 2C: создание блока процесса ядра

На этом подэтапе работы *CreateProcess* инициализируется блок KPROCESS, содержащий указатель на список потоков ядра. (Ядро не имеет представления об описателях, поэтому оно обходит их таблицу.) Блок процесса ядра также указывает на каталог таблицы страниц процесса (используемый для отслеживания виртуального адресного пространства процесса) и содержит суммарное время выполнения потоков процесса, базовый приоритет процесса по умолчанию (он начинается с Normal, или 8, если только его значение у родительского процесса не равно Idle или Below Normal; в последнем случае приоритет просто наследуется), привязку потоков к процессорам по умолчанию и начальный квант процессорного времени, выделяемый процессу по умолчанию. Последнее значение принимается равным *PspForegroundQuantum[0]*, первому элементу общесистемной таблицы величин квантов.

ПРИМЕЧАНИЕ Начальный квант по умолчанию в клиентских и серверных версиях Windows неодинаков. Подробнее о квантах см. раздел «Планирование потоков» далее в этой главе.

Этап 2D: инициализация адресного пространства процесса

Подготовка адресного пространства нового процесса довольно сложна, поэтому разберем ее отдельно по каждой операции. Для максимального усвоения материала этого раздела вы должны иметь представление о внутреннем устройстве диспетчера памяти Windows (см. главу 7).

- Диспетчер виртуальной памяти присваивает времени последнего усечения (last trim time) для процесса текущее время. Диспетчер рабочих наборов, выполняемый в контексте системного потока диспетчера настройки баланса (balance set manager), использует это значение, чтобы определить, когда нужно инициировать усечение рабочего набора.
- Диспетчер памяти инициализирует список рабочего набора процесса, после чего становится возможной генерация ошибок страниц.
- Раздел (созданный при открытии файла образа) проецируется на адресное пространство нового процесса, и базовый адрес раздела процесса приравнивается базовому адресу образа.
- На адресное пространство процесса проецируется Ntdll.dll.
- На адресное пространство процесса проецируются общесистемные таблицы NLS (national language support).

ПРИМЕЧАНИЕ Процессы POSIX клонируют адресное пространство своих родителей, поэтому для них не нужны все вышеперечисленные операции создания нового адресного пространства. В случае приложений POSIX базовый адрес раздела нового процесса приравнивается тому же базовому адресу родительского процесса, а родительский PEB просто копируется.

Этап 2E: формирование блока PEB

CreateProcess выделяет страницу под PEB и инициализирует некоторые поля, описанные в таблице 6-6.

Таблица 6-6. *Начальные значения полей PEB*

Поле	Начальное значение
ImageBaseAddress	Базовый адрес раздела
NumberOfProcessors	Значение переменной ядра *KeNumberProcessors*
NtGlobalFlag	Значение переменной ядра *NtGlobalFlag*
CriticalSectionTimeout	Значение переменной ядра *MmCriticalSectionTimeout*
HeapSegmentReserve	Значение переменной ядра *MmHeapSegmentReserve*
HeapSegmentCommit	Значение переменной ядра *MmHeapSegmentCommit*
HeapDeCommitTotalFree-Threshold	Значение переменной ядра *MmHeapDeCommitTotal-FreeThreshold*

Таблица 6-6. (*окончание*)

Поле	Начальное значение
HeapDeCommitFreeBlock-Threshold	Значение переменной ядра *MmHeapDeCommitFreeBlock-Threshold*
NumberOfHeaps	0
MaximumNumberOfHeaps	(Размер страницы – размер PEB) / 4
ProcessHeaps	Первый байт после PEB
OSMajorVersion	Значение переменной ядра *NtMajorVersion*
OSMinorVersion	Значение переменной ядра *NtMinorVersion*
OSBuildNumber	Значение переменной ядра *NtBuildNumber* & 0x3FFF
OSPlatformId	2

Если в файле явно указаны значения версии Windows, эти данные замещают соответствующие начальные значения, показанные в таблице 6-6. Связь полей версии из заголовка образа с полями PEB описывается в таблице 6-7.

Таблица 6-7. *Windows-значения, заменяющие начальные значения полей PEB*

Имя поля	Значение из заголовка образа
OSMajorVersion	НеобязательныйЗаголовок.НомерВерсииWin32 & 0xFF
OSMinorVersion	(НеобязательныйЗаголовок.НомерВерсииWin32 >> 8) & 0xFF
OSBuildNumber	(НеобязательныйЗаголовок.НомерВерсииWin32 >> 16) & 0x3FFF
OSPlatformId	(НеобязательныйЗаголовок.НомерВерсииWin32 >> 30) ^ 0x2

Этап 2F: завершение инициализации объекта «процесс» исполнительной системы

Перед возвратом описателя нового процесса выполняется несколько завершающих операций.

1. Если общесистемный аудит процессов разрешен (через локальную политику безопасности или политику группы, вводимую контроллером домена), факт создания процесса отмечается в журнале безопасности.
2. Если родительский процесс входил в задание, новый процесс тоже включается в это задание (о заданиях — в конце главы).
3. Если в заголовке образа задан флаг IMAGE_FILE_UP_SYSTEM_ONLY (который указывает, что данную программу можно запускать только в однопроцессорной системе), для выполнения всех потоков процесса выбирается один процессор. Выбор осуществляется простым перебором доступных процессоров: при каждом запуске следующей программы такого типа выбирается следующий процессор. Благодаря этому подобные программы равномерно распределяются между процессорами.
4. Если в образе явно указана маска привязки к процессорам (например, в поле конфигурационного заголовка), ее значение копируется в PEB и впоследствии устанавливается как маска привязки к процессорам по умолчанию.

5. *CreateProcess* помещает блок нового процесса в конец списка активных процессов (*PsActiveProcessHead*).
6. Устанавливается время создания процесса, и вызвавшей функции (*CreateProcess* в Kernel32.dll) возвращается описатель нового процесса.

Этап 3: создание первичного потока, его стека и контекста

К началу третьего этапа объект «процесс» исполнительной системы полностью инициализирован. Однако у него еще нет ни одного потока, поэтому он не может ничего делать. Прежде чем создать поток, нужно создать стек и контекст, в котором он будет выполняться. Эта операция и является целью данного этапа. Размер стека первичного потока берется из образа — другого способа задать его размер нет.

Далее создается первичный поток вызовом *NtCreateThread*. Параметр потока — это адрес PEB (данный параметр нельзя задать при вызове *CreateProcess* — только при вызове *CreateThread*). Этот параметр используется кодом инициализации, выполняемым в контексте нового потока (см. этап 6). Однако поток по-прежнему ничего не делает — он создается в приостановленном состоянии и возобновляется лишь по завершении инициализации процесса (см. этап 5). *NtCreateThread* вызывает *PspCreateThread* (функцию, которая используется и при создании системных потоков) и выполняет следующие операции.

1. Увеличивается счетчик потоков в объекте «процесс».
2. Создается и инициализируется блок потока исполнительной системы (ETHREAD).
3. Генерируется идентификатор нового потока.
4. В адресном пространстве пользовательского режима формируется TEB.
5. Стартовый адрес потока пользовательского режима сохраняется в блоке ETHREAD. В случае Windows-потоков это адрес системной стартовой функции потока в Kernel32.dll (*BaseProcessStart* для первого потока в процессе и *BaseThreadStart* для дополнительных потоков). Стартовый адрес, указанный пользователем, также хранится в ETHREAD, но в другом его месте; это позволяет системной стартовой функции потока вызвать пользовательскую стартовую функцию.
6. Для подготовки блока KTHREAD вызывается *KeInitThread*. Начальный и текущий базовые приоритеты потока устанавливаются равными базовому приоритету процесса; привязка к процессорам и значение кванта также устанавливаются по соответствующим параметрам процесса. Кроме того, функция определяет идеальный процессор для первичного потока. (О том, как происходит выбор идеального процессора см. раздел «Идеальный и последний процессоры» далее в этой главе.) Затем *KeInitThread* создает стек ядра для потока и инициализирует его аппаратно-зависимый контекст, включая фреймы ловушек и исключений. Контекст потока настраивается так, чтобы выполнение этого потока началось в режиме ядра

в *KiThreadStartup*. Далее *KeInitThread* устанавливает состояние потока в Initialized (инициализирован) и возвращает управление *PspCreateThread*.

7. Вызываются общесистемные процедуры, зарегистрированные на уведомление о создании потока.
8. Маркер доступа потока настраивается как указатель на маркер доступа процесса. Затем вызывающая программа проверяется на предмет того, имеет ли она право создавать потоки. Эта проверка всегда заканчивается успешно, если поток создается в локальном процессе, но может дать отрицательный результат, если поток создается в другом процессе через функцию *CreateRemoteThread* и у создающего процесса нет привилегии отладки.
9. Наконец, поток готов к выполнению.

Этап 4: уведомление подсистемы Windows о новом процессе

Если заданы соответствующие правила, для нового процесса создается маркер с ограниченными правами доступа. К этому моменту все необходимые объекты исполнительной системы созданы, и Kernel32.dll посылает подсистеме Windows сообщение, чтобы она подготовилась к выполнению нового процесса и потока. Сообщение включает следующую информацию:

- описатели процесса и потока;
- флаги создания;
- идентификатор родительского процесса;
- флаг, который указывает, относится ли данный процесс к Windows-приложениям (что позволяет Csrss определить, показывать ли курсор запуска).

Получив такое сообщение, подсистема Windows выполняет следующие операции.

1. *CreateProcess* дублирует описатели процесса и потока. На этом этапе счетчик числа пользователей процесса увеличивается с 1 (начального значения, установленного в момент создания процесса) до 2.
2. Если класс приоритета процесса не указан, *CreateProcess* устанавливает его в соответствии с алгоритмом, описанным ранее.
3. Создается блок процесса Csrss.
4. Порт исключений нового процесса настраивается как общий порт функций для подсистемы Windows, которая может таким образом получать сообщения при возникновении в процессе исключений (об обработке исключений см. главу 3).
5. Если в данный момент процесс отлаживается (т. е. подключен к процессу отладчика), в качестве общего порта функций выбирается отладочный порт. Такой вариант позволяет Windows пересылать события отладки в новом процессе (генерируемые при создании и удалении потоков, при исключениях и т. д.) в виде сообщений подсистеме Windows, которая

затем доставляет их процессу, выступающему в роли отладчика нового процесса.

6. Создается и инициализируется блок потока Csrss.
7. *CreateProcess* включает поток в список потоков процесса.
8. Увеличивается счетчик процессов в данном сеансе.
9. Уровень завершения процесса (process shutdown level) устанавливается как 0x280 (это значение по умолчанию; его описание ищите в документации MSDN Library по ключевому слову *SetProcessShutdownParameters*).
10. Блок нового процесса включается в список общесистемных Windows-процессов.
11. Создается и инициализируется структура данных (W32PROCESS), индивидуальная для каждого процесса и используемая той частью подсистемы Windows, которая работает в режиме ядра.
12. Выводится курсор запуска в виде стрелки с песочными часами. Тем самым Windows говорит пользователю: «Я запускаю какую-то программу, но ты все равно можешь пользоваться курсором.» Если в течение двух секунд процесс не делает GUI-вызова, курсор возвращается к стандартному виду. А если за это время процесс обратился к GUI, *CreateProcess* ждет открытия им окна в течение пяти секунд и после этого восстанавливает исходную форму курсора.

Этап 5: запуск первичного потока

К началу этого этапа окружение процесса уже определено, его потокам выделены ресурсы, у процесса есть поток, а подсистеме Windows известен факт существования нового процесса. Поэтому для завершения инициализации нового процесса (см. этап 6) возобновляется выполнение его первичного потока, если только не указан флаг CREATE_SUSPENDED.

Этап 6: инициализация в контексте нового процесса

Новый поток начинает свою жизнь с выполнения стартовой процедуры потока режима ядра, *KiThreadStartup*, которая понижает уровень IRQL потока с «DPC/dispatch» до «APC», а затем вызывает системную стартовую процедуру потока, *PspUserThreadStartup*. Параметром этой процедуры является пользовательский стартовый адрес потока.

В Windows 2000 *PspUserThreadStartup* сначала разрешает расширение рабочего набора. Если создаваемый процесс является отлаживаемой программой, все его потоки (которые могли быть созданы на этапе 3) приостанавливаются. В отладочный порт процесса (порт функций подсистемы Windows, так как это Windows-процесс) посылается сообщение о создании процесса, чтобы подсистема доставила событие отладки CREATE_PROCESS_DEBUG_INFO соответствующему отладчику. Далее *PspUserThreadStartup* ждет пересылки подсистемой Windows ответа отладчика (через функцию *Conti-*

nueDebugEvent). Как только такой ответ приходит, выполнение всех потоков возобновляется.

В Windows XP и Windows Server 2003 *PspUserThreadStartup* проверяет, разрешена ли в системе предварительная выборка для приложений (application prefetching), и, если да, вызывает модуль логической предвыборки (logical prefetcher) для обработки файла команд предвыборки (prefetch instruction file) (если таковой есть), а затем выполняет предвыборку страниц, на которые процесс ссылался в течение первых десяти секунд при последнем запуске. Наконец, *PspUserThreadStartup* ставит APC пользовательского режима в очередь для запуска процедуры инициализации загрузчика образов (*LdrInitializeThunk* из Ntdll.dll). APC будет доставлен, когда поток попытается вернуться в пользовательский режим.

Когда *PspUserThreadStartup* возвращает управление *KiThreadStartup*, та возвращается из режима ядра, доставляет APC и обращается к *LdrInitializeThunk*. Последняя инициализирует загрузчик, диспетчер кучи, таблицы NLS, массив локальной памяти потока (thread-local storage, TLS) и структуры критической секции. После этого она загружает необходимые DLL и вызывает их точки входа с флагом DLL_PROCESS_ATTACH.

Наконец, когда процедура инициализации загрузчика возвращает управление диспетчеру APC пользовательского режима, начинается выполнение образа в пользовательском режиме. Диспетчер APC вызывает стартовую функцию потока, помещенную в пользовательский стек в момент доставки APC.

Сборки, существующие в нескольких версиях

Одна из проблем, уже давно изводившая пользователей Windows, — так называемый «DLL hell». Вы создаете этот ад, устанавливая приложение, которое заменяет одну или более базовых системных DLL, содержащих, например, стандартные элементы управления, исполняющую среду Microsoft Visual Basic или MFC. Программы установки приложений делают такую замену, чтобы приложения работали корректно, но обновленные DLL могут оказаться несовместимыми с уже установленными приложениями.

Эта проблема в Windows 2000 была отчасти решена, где модификация базовых системных DLL предотвращалась средством Windows File Protection, а приложениям разрешалось использовать собственные экземпляры этих DLL. Чтобы задействовать свой экземпляр какой-либо DLL вместо того, который находится в системном каталоге, у приложения должен быть файл *Application*.exe.local (где *Application* — имя исполняемого файла приложения); этот файл указывает загрузчику сначала проверить DLL-модули в каталоге приложения. Такой вид переадресации DLL позволяет избежать проблем несовместимости между приложениями и DLL, но больно бьет по принципу разделения DLL, ради которого DLL изначально и разрабатывались. Кроме того, любые DLL, загруженные из списка KnownDLLs (они постоянно проецируются

см. след. стр.

в память) или, наоборот, загруженные ими, нельзя переадресовывать по такому механизму.

Продолжая работу над решением этой проблемы, Microsoft ввела в Windows XP общие сборки (shared assemblies). Сборка (assembly) состоит из группы ресурсов, в том числе DLL и XML-файла манифеста, который описывает сборку и ее содержимое. Приложение ссылается на сборку через свой XML-манифест. Манифестом может быть файл в каталоге приложения с тем же именем, что и само приложение, но с добавленным расширением «.manifest» (например application.exe.manifest), либо он может быть включен в приложение как ресурс. Манифест описывает приложение и его зависимости от сборок.

Существует два типа сборок: закрытые (private) и общие (shared). Общие сборки отличаются тем, что они подписываются цифровой подписью; это позволяет обнаруживать их повреждение или модификацию. Помимо этого, общие сборки хранятся в каталоге \Windows\Winsxs, тогда как закрытые — в каталоге приложения. Таким образом, с общими сборками сопоставлен файл каталога (.cat), содержащий информацию о цифровых подписях. Общие сборки могут содержать несколько версий какой-либо DLL, чтобы приложения, зависимые от определенной версии этой DLL, всегда могли использовать именно ее.

Обычно файлу манифеста сборки присваивается имя, которое включает имя сборки, информацию о версии, некий текст, представляющий уникальную сигнатуру, и расширение .manifest. Манифесты хранятся в каталоге \Windows\Winsxs\Manifests, а остальные ресурсы сборки — в подкаталогах \Windows\Winsxs с теми же именами, что и у соответствующих файлов манифестов, но без расширения .manifest.

Пример общей сборки — 6-я версия DLL стандартных элементов управления Windows, comctl32.dll, которая является новинкой Windows XP. Ее файл манифеста называется \Windows\Winsxs\Manifest\x86_Microsoft.Windows.CommonControls_6595b64144ccf1df_6.0.0.0_x-ww_1382d70a.manifest. С ним сопоставлен файл каталога (с тем же именем, но с расширением .cat) и подкаталог в Winsxs, включающий comctl32.dll.

Comctl32.dll версии 6 обеспечивает интеграцию с темами Windows XP и из-за того, что приложения, написанные без учета поддержки тем, могут неправильно выглядеть на экране при использовании этой новой DLL, она доступна только тем приложениям, которые явно ссылаются на ее общую сборку. Версия Comctl32.dll, установленная в \Windows\System32, — это экземпляр версии 5.x, не поддерживающей темы. Загружая приложение, загрузчик ищет его манифест и, если таковой есть, загружает DLL-модули из указанной сборки. DLL, не включенные в сборки, на которые ссылается манифест, загружаются традиционным способом. Поэтому унаследованные приложения связываются с версией в \Windows\System32, а новые приложения с поддержкой тем могут ссылаться на новую версию в своих манифестах.

Чтобы увидеть, какой эффект дает манифест, указывающий системе задействовать новую библиотеку стандартных элементов управления в Windows XP, запустите User State Migration Wizard (\Windows\System32\Usmt\Migwiz.exe) с файлом манифеста и без него.

1. Запустите этот мастер и обратите внимание на темы Windows XP на кнопках в мастере.
2. Откройте файл манифеста в Notepad и найдите упоминание 6-й версии библиотеки стандартных элементов управления.
3. Переименуйте Migwiz.exe.manifest в Migwiz.exe.manifest.bak.
4. Вновь запустите мастер и обратите внимание на кнопки без тем.
5. Восстановите исходное имя файла манифеста.

И еще одно преимущество общих сборок. Издатель может указать конфигурацию, которая заставит все приложения, использующие определенную сборку, работать с ее обновленной версией. Издатели поступают так, когда хотят сохранить обратную совместимость, пока занимаются устранением каких-то ошибок. Однако благодаря гибкости модели сборок приложение может игнорировать новые настройки и по-прежнему использовать более старую версию.

Внутреннее устройство потоков

Теперь, изучив анатомию процессов, рассмотрим структуру потоков. Там, где явно не сказано обратное, считайте, что весь материал этого раздела в равной мере относится как к обычным потокам пользовательского режима, так и к системным потокам режима ядра (описанным в главе 3).

Структуры данных

На уровне операционной системы поток представляется блоком потока, принадлежащим исполнительной системе (ETHREAD). Структура этого блока показана на рис. 6-7. Блок ETHREAD и все структуры данных, на которые он ссылается, существуют в системном адресном пространстве, кроме блока переменных окружения потока (thread environment block, TEB) — он размещается в адресном пространстве процесса. Помимо этого, процесс подсистемы Windows (Csrss) поддерживает параллельную структуру для каждого потока, созданного в Windows-процессе. Часть подсистемы Windows, работающая в режиме ядра (Win32k.sys), также поддерживает для каждого потока, вызывавшего USER- или GDI-функцию, структуру W32THREAD, на которую указывает блок ETHREAD.

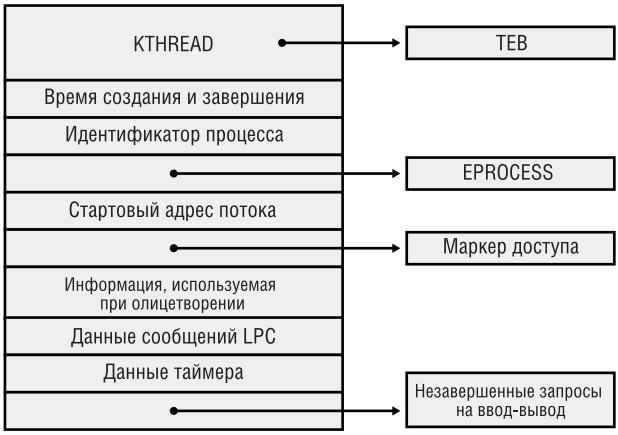


Рис. 6-7. *Блок потока исполнительной системы*

Поля блока потока, показанные на рис. 6-7, в большинстве своем не требуют дополнительных пояснений. Первое поле — это блок потока ядра (KTHREAD). За ним следуют идентификационные данные потока и процесса (включая указатель на процесс — владелец данного потока, что обеспечивает доступ к информации о его окружении), затем информация о защите в виде указателя на маркер доступа и сведения, необходимые для олицетворения (подмены одного процесса другим), а также поля, связанные с сообщениями LPC и незавершенными запросами на ввод-вывод. В таблице 6-8 даны ссылки на другие части книги, где некоторые из наиболее важных полей описываются подробнее. Чтобы получить более детальные сведения о внутренней структуре блока ETHREAD, используйте команду *dt* отладчика ядра.

Таблица 6-8. *Ключевые поля блока потока*

Элемент	Описание	Дополнительная ссылка
KTHREAD	См. таблицу 6-9	
Временные показатели потока	Время создания и завершения потока	
Идентификационные данные процесса	Идентификатор процесса и указатель на блок EPROCESS процесса, которому принадлежит данный поток	
Стартовый адрес	Адрес стартовой процедуры потока	
Информация об олицетворении	Маркер доступа и уровень олицетворения (если поток олицетворяет, или подменяет, клиент)	Глава 8
Информация LPC	Идентификатор и адрес сообщения, ожидаемого потоком	Раздел «LPC» главы 3
Информация, связанная с вводом-выводом	Список необработанных пакетов запросов на ввод-вывод (I/O request packets, IRP)	Глава 9

Давайте повнимательнее присмотримся к двум ключевым структурам потока, упомянутым выше, — к блокам KTHREAD и TEB. Первый содержит информацию, нужную ядру Windows для планирования потоков и их синхронизации с другими потоками. Схема блока KTHREAD показана на рис. 6-8.

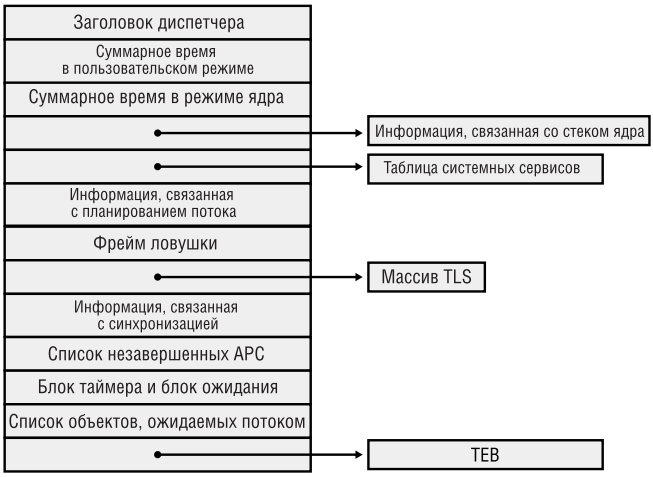


Рис. 6-8. *Схема блока потока ядра*

Ключевые поля блока KTHREAD кратко рассмотрены в таблице 6-9.

Таблица 6-9. *Ключевые поля блока KTHREAD*

Элемент	Описание	Дополнительная ссылка
Заголовок диспетчера	Поскольку поток — это объект, на котором можно ждать, он запускается со стандартным заголовком объекта диспетчера ядра	Раздел «Объекты диспетчера ядра» главы 3
Время выполнения	Суммарное время выполнения (в режиме ядра и в пользовательском режиме)	
Указатель на информацию стека ядра	Базовый и верхний адреса стека ядра	Глава 7
Указатель на таблицу системных сервисов	В начале выполнения каждого потока указатель в данном поле ссылается на главную таблицу системных сервисов (*KeServiceDescriptorTable*); когда поток впервые вызывает GUI-сервис Windows, текущей таблицей системных сервисов становится та, которая включает сервисы GDI и USER в Win32k.sys	Раздел «Диспетчеризация системных сервисов» главы 3
Информация, связанная с планированием	Базовый и текущий приоритеты, значение кванта, маска привязки к процессорам, идеальный процессор, статус планирования, счетчики числа замораживаний (freeze count) и приостановок (suspend count)	Раздел «Планирование потоков» далее в этой главе

см. след. стр.

Таблица 6-9. (*окончание*)

Элемент	Описание	Дополнительная ссылка
Блоки ожидания	В блоке потока содержится четыре встроенных блока ожидания, поэтому их не нужно создавать и инициализировать при каждом переходе потока в состояние ожидания какого-либо объекта (один блок ожидания принадлежит таймерам)	Раздел «Синхронизация» главы 3
Информация об ожидании	Список объектов, ожидаемых потоком, причина ожидания и время перехода потока в состояние ожидания	Раздел «Синхронизация» главы 3
Список мутантов	Список принадлежащих потоку объектов «мутант»	Раздел «Синхронизация» главы 3
Очереди APC	Список незавершенных APC режима ядра и пользовательского режима, а также флаг оповещения (alertable flag)	Раздел «APC» главы 3
Блок таймера	Встроенный блок таймера (а также соответствующий блок ожидания)	
Список очередей	Указатель на сопоставленный с потоком объект очереди	Раздел «Синхронизация» главы 3
Указатель на TEB	Идентификатор потока, данные TLS, указатель на PEB, информация, связанная с GDI и OpenGL	

ЭКСПЕРИМЕНТ: просмотр структур ETHREAD и KTHREAD

Структуры ETHREAD и KTHREAD можно просмотреть с помощью команды *dt* отладчика ядра. В следующем выводе показан формат ETHREAD:

```
lkd> dt nt!_ethread
nt!_ETHREAD
   +0x000 Tcb              : _KTHREAD
   +0x1c0 CreateTime       : _LARGE_INTEGER
   +0x1c0 NestedFaultCount : Pos 0, 2 Bits
   +0x1c0 ApcNeeded        : Pos 2, 1 Bit
   +0x1c8 ExitTime         : _LARGE_INTEGER
   +0x1c8 LpcReplyChain    : _LIST_ENTRY
   +0x1c8 KeyedWaitChain   : _LIST_ENTRY
   +0x1d0 ExitStatus       : Int4B
   +0x1d0 OfsChain         : Ptr32 Void
   +0x1d4 PostBlockList    : _LIST_ENTRY
   +0x1dc TerminationPort  : Ptr32 _TERMINATION_PORT
   +0x1dc ReaperLink       : Ptr32 _ETHREAD
   +0x1dc KeyedWaitValue   : Ptr32 Void
   +0x1e0 ActiveTimerListLock : Uint4B
   +0x1e4 ActiveTimerListHead : _LIST_ENTRY
   +0x1ec Cid              : _CLIENT_ID
   +0x1f4 LpcReplySemaphore : _KSEMAPHORE
```

```
   +0x1f4 KeyedWaitSemaphore : _KSEMAPHORE
   +0x208 LpcReplyMessage  : Ptr32 Void
   +0x208 LpcWaitingOnPort : Ptr32 Void
   +0x20c ImpersonationInfo : Ptr32 _PS_IMPERSONATION_INFORMATION
   +0x210 IrpList          : _LIST_ENTRY
   +0x218 TopLevelIrp      : Uint4B
   +0x21c DeviceToVerify   : Ptr32 _DEVICE_OBJECT
   +0x220 ThreadsProcess   : Ptr32 _EPROCESS
   +0x224 StartAddress     : Ptr32 Void
   +0x228 Win32StartAddress : Ptr32 Void
   +0x228 LpcReceivedMessageId : Uint4B
   +0x22c ThreadListEntry  : _LIST_ENTRY
   +0x234 RundownProtect   : _EX_RUNDOWN_REF
   +0x238 ThreadLock       : _EX_PUSH_LOCK
   +0x23c LpcReplyMessageId : Uint4B
   +0x240 ReadClusterSize  : Uint4B
   +0x244 GrantedAccess    : Uint4B
   +0x248 CrossThreadFlags : Uint4B
   +0x248 Terminated       : Pos 0, 1 Bit
   +0x248 DeadThread       : Pos 1, 1 Bit
   +0x248 HideFromDebugger : Pos 2, 1 Bit
   +0x248 ActiveImpersonationInfo : Pos 3, 1 Bit
   +0x248 SystemThread     : Pos 4, 1 Bit
   +0x248 HardErrorsAreDisabled: Pos 5, 1 Bit
   +0x248 BreakOnTermination : Pos 6, 1 Bit
   +0x248 SkipCreationMsg  : Pos 7, 1 Bit
   +0x248 SkipTerminationMsg : Pos 8, 1 Bit
   +0x24c SameThreadPassiveFlags : Uint4B
   +0x24c ActiveExWorker   : Pos 0, 1 Bit
   +0x24c ExWorkerCanWaitUser : Pos 1, 1 Bit
   +0x24c MemoryMaker      : Pos 2, 1 Bit
   +0x250 SameThreadApcFlags : Uint4B
   +0x250 LpcReceivedMsgIdValid: Pos 0, 1 Bit
   +0x250 LpcExitThreadCalled : Pos 1, 1 Bit
   +0x250 AddressSpaceOwner : Pos 2, 1 Bit
   +0x254 ForwardClusterOnly : UChar
   +0x255 DisablePageFaultClustering : UChar
```

Для просмотра KTHREAD предназначена аналогичная команда:

```
lkd> dt nt!_kthread
nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 MutantListHead   : _LIST_ENTRY
   +0x018 InitialStack     : Ptr32 Void
   +0x01c StackLimit       : Ptr32 Void
   +0x020 Teb              : Ptr32 Void
   +0x024 TlsArray         : Ptr32 Void
   +0x028 KernelStack      : Ptr32 Void
   +0x02c DebugActive      : UChar
```

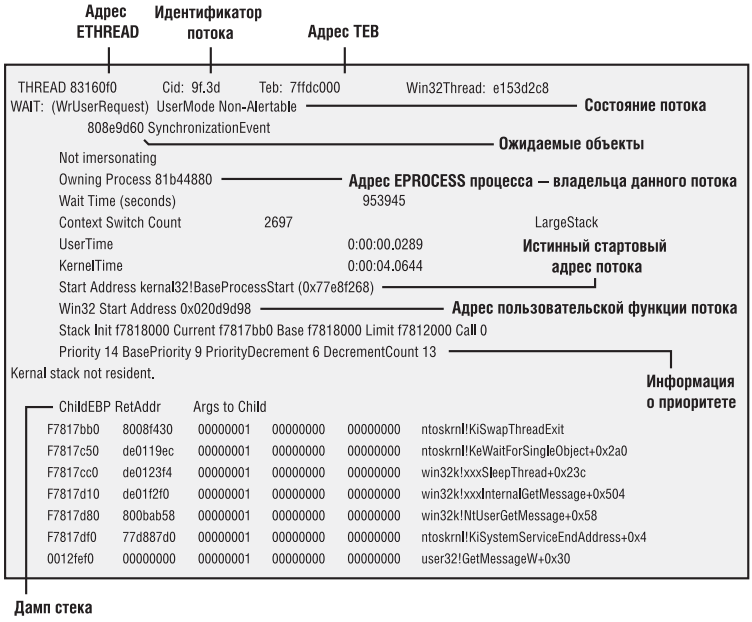
см. след. стр.

```
+0x02d State            : UChar
+0x02e Alerted          : [2] UChar
+0x030 Iopl             : UChar
+0x031 NpxState         : UChar
+0x032 Saturation       : Char
+0x033 Priority         : Char
+0x034 ApcState         : _KAPC_STATE
+0x04c ContextSwitches  : Uint4B
+0x050 IdleSwapBlock    : UChar
+0x051 Spare0           : [3] UChar
+0x054 WaitStatus       : Int4B
```

ЭКСПЕРИМЕНТ: использование команды *!thread* отладчика ядра

Команда *!thread* отладчика ядра выдает дамп подмножества информации из структур данных потока. Отладчик ядра выводит ряд важных данных, не показываемых любыми другими утилитами: адреса внутренних структур, детальные сведения о приоритетах, данные стека, список незавершенных запросов на ввод-вывод и список ожидаемых объектов для тех потоков, которые находятся в состоянии ожидания.

Чтобы получить информацию о потоке, используйте либо команду *!process* (которая выводит все блоки потоков после блока процесса), либо команду *!thread* (которая сообщает сведения только об указанном потоке). Ниже дан пример информации о потоке с пояснением ее важнейших полей.



ЭКСПЕРИМЕНТ: просмотр информации о потоке

Утилита Tlist из Windows Debugging Tools позволяет получить подробную информацию о процессе, пример которой приведен ниже. Заметьте, что в списке потоков указывается «Win32StartAddress». Это адрес, передаваемый функции *CreateThread* приложением. Остальные утилиты, кроме Process Explorer, показывающие стартовый адрес потока, выводят его истинный стартовый адрес, а не стартовый адрес, заданный приложением.

```
C:\> tlist winword
 155 WINWORD.EXE       Document1 - Microsoft Word
   CWD:     C:\book\
   CmdLine: "C:\Program Files\Microsoft Office\Office\WINWORD.EXE"
   VirtualSize:    64448 KB   PeakVirtualSize:   106748 KB
   WorkingSetSize:  1104 KB   PeakWorkingSetSize:  6776 KB
   NumberOfThreads: 2
    156 Win32StartAddr:0x5032cfdb LastErr:0x00000000 State:Waiting
    167 Win32StartAddr:0x00022982 LastErr:0x00000000 State:Waiting
                     0x50000000  WINWORD.EXE
     5.0.2163.1 shp  0x77f60000  ntdll.dll
     5.0.2191.1 shp  0x77f00000  KERNEL32.dll
         ... список DLL, загруженных в процесс...
```

В отличие от других структур данных, описываемых в этом разделе, только блок TEB, показанный на рис. 6-9, присутствует в адресном пространстве процесса, а не системы. В TEB хранится контекстная информация загрузчика образов и различных Windows DLL. Поскольку эти компоненты выполняются в пользовательском режиме, им нужны структуры данных, доступные для записи из этого режима. Вот почему TEB размещается в адресном пространстве процесса, а не системы, где он был бы доступен для записи только из режима ядра. Адрес TEB можно найти с помощью команды *!thread* отладчика ядра.

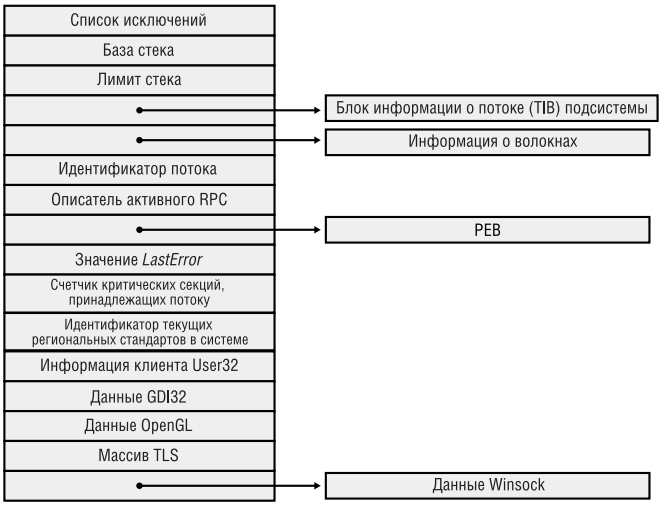


Рис. 6-9. *Поля блока переменных окружения потока*

ЭКСПЕРИМЕНТ: исследуем TEB

Вы можете получить дамп структуры TEB, используя команду *!teb* отладчика ядра. Ее вывод выглядит так:

```
kd> !teb
TEB at 7ffde000
    ExceptionList:        0006b540
    StackBase:            00070000
    StackLimit:           00065000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7ffde000
    EnvironmentPointer:   00000000
    ClientId:             00000254 . 000007ac
    RpcHandle:            00000000
    Tls Storage:          00000000
    PEB Address:          7ffdf000
    LastErrorValue:       2
    LastStatusValue:      c0000034
    Count Owned Locks:    0     HardErrorMode:        0
```

Переменные ядра

Как и в случае процессов, ряд переменных ядра Windows контролирует выполнение потоков. Список таких переменных, связанных с потоками, приводится в таблице 6-10.

Таблица 6-10. *Переменные ядра, относящиеся к потокам*

Переменная	Тип	Описание
PspCreateThread-NotifyRoutine	Массив указателей	Массив указателей на процедуры (максимум 8), вызываемых при создании и удалении потока
PspCreateThread-NotifyRoutineCount	DWORD	Счетчик зарегистрированных процедур уведомления потока
PspCreateProcess-NotifyRoutine	Массив указателей	Массив указателей на процедуры (максимум 8), вызываемых при создании и удалении процесса

Счетчики производительности

Большая часть важной информации в структурах данных потоков экспортируется в виде счетчиков производительности, перечисленных в таблице 6-11. Даже используя только оснастку Performance, вы можете получить довольно много сведений о внутреннем устройстве потоков.

Таблица 6-11. *Счетчики производительности для потоков*

Объект: счетчик	Описание
Process: Priority Base (Процесс: Базовый приоритет)	Возвращает текущий базовый приоритет процесса; это начальный приоритет для потоков, создаваемых в данном процессе
Thread: % Privileged Time (Поток: % работы в привилегированном режиме)	Процентная доля времени, в течение которого поток выполнялся в режиме ядра
Thread: % Processor Time (Поток: % загруженности процессора)	Процентная доля процессорного времени, использованная потоком за определенный период времени; вычисляется как сумма % Privileged Time и % User Time
Thread: % User Time (Поток: % работы в пользовательском режиме	Процентная доля времени, в течение которого поток выполнялся в пользовательском режиме
Thread: Context Switches/Sec (Поток: Контекстных переключений/сек)	Число переключений контекстов в секунду в системе
Thread: Elapsed Time (Поток: Прошло времени)	Процессорное время (в секундах), использованное потоком
Thread: ID Process (Поток: Идентификатор процесса)	Идентификатор процесса — владельца потока; действителен лишь на время существования процесса, так как идентификаторы процессов подлежат повторному использованию
Thread: ID Thread (Поток: Идентификатор потока)	Идентификатор потока; действителен лишь на время существования потока, так как идентификаторы потоков подлежат повторному использованию
Thread: Priority Base (Поток: Базовый приоритет)	Текущий базовый приоритет потока; его значение может отличаться от начального базового приоритета
Thread: Priority Current (Поток: Текущий приоритет)	Текущий динамический приоритет потока
Thread: Start Address (Поток: Начальный адрес)	Стартовый виртуальный адрес потока (у большинства потоков этот адрес одинаков)
Thread: Thread State (Поток: Состояние потока)	Текущее состояние потока — значение в интервале от 0 до 7
Thread: Thread Wait Reason (Поток: Причина состояния ожидания для потока)	Причина перехода потока в состояние ожидания — значение в интервале от 0 до 19

Сопутствующие функции

В таблице 6-12 перечислены Windows-функции, позволяющие создавать потоки и манипулировать ими. Здесь не показаны функции, связанные с планированием и управлением приоритетами потоков, — они описываются в разделе «Планирование потоков» далее в этой главе.

Таблица 6-12. *Функции Windows, относящиеся к потокам*

Функция	Описание
CreateThread	Создает новый поток
CreateRemoteThread	Создает поток в другом процессе
ExitThread	Нормально завершает поток
TerminateThread	Аварийно завершает поток
GetExitCodeThread	Получает код завершения другого потока
GetThreadTimes	Возвращает временные характеристики другого потока
GetCurrentThread	Возвращает псевдоописатель текущего потока
GetCurrentThreadId	Возвращает идентификатор текущего потока
GetThreadId	Возвращает идентификатор указанного потока
GetThreadContext *SetThreadContext*	Возвращает или изменяет регистры процессора для данного потока
GetThreadSelectorEntry	Возвращает элемент таблицы дескрипторов другого потока (только для систем типа x86)

Рождение потока

Жизненный цикл потока начинается при его создании программой. Запрос на его создание в конечном счете поступает исполнительной системе Windows, где диспетчер процессов выделяет память для объекта «поток» и вызывает ядро для инициализации блока потока ядра. Ниже перечислены основные этапы создания потока Windows-функцией *CreateThread* (которая находится в Kernel32.dll).

1. *CreateThread* создает стек пользовательского режима в адресном пространстве процесса.
2. *CreateThread* инициализирует аппаратный контекст потока, специфичный для конкретной архитектуры процессора. (Подробнее о блоке контекста потока см. раздел справочной документации Windows API по структуре CONTEXT.)
3. Для создания объекта «поток» исполнительной системы вызывается *NtCreateThread*. Он создается в приостановленном состоянии. Описание операций, выполняемых *NtCreateThread*, см. в разделе «Что делает функция *CreateProcess*» (этапы 3 и 6) ранее в этой главе.
4. *CreateThread* уведомляет подсистему Windows о создании нового потока, и та выполняет некоторые подготовительные операции.
5. Вызвавшему коду возвращаются описатель и идентификатор потока (сгенерированный на этапе 3).
6. Выполнение потока возобновляется, и ему может быть выделено процессорное время, если только он не был создан с флагом CREATE_SUSPENDED. Перед вызовом по пользовательскому стартовому адресу поток выполняет операции, описанные в разделе «Этап 3: создание первичного потока, его стека и контекста» ранее в этой главе.

Наблюдение за активностью потоков

Не только оснастка Performance, но и другие утилиты (таблица 6-13) позволяют получать сведения о состоянии потоков в Windows. (Утилиты, показывающие информацию о планировании потоков, перечисляются в разделе «Планирование потоков» далее в этой главе.)

ПРИМЕЧАНИЕ Чтобы получить информацию о потоке с помощью Tlist, введите *tlist xxx*, где *xxx* — имя образа процесса или заголовок окна (можно использовать символы подстановки).

Таблица 6-13. *Утилиты для исследования потоков и их функций*

Объект	Perfmon	Pviewer	Pstat	Qslice	Tlist	KD *!thread*	Process Explorer	Pslist
ID потока	✓	✓	✓		✓	✓	✓	✓
Истинный стартовый адрес	✓	✓	✓			✓	✓	✓
Стартовый адрес Win32					✓	✓	✓	
Текущий адрес	✓	✓				✓	✓	
Число переключений контекста	✓	✓	✓				✓	✓
Общее время работы в пользовательском режиме		✓	✓			✓	✓	✓
Общее время работы в привилегированном режиме		✓	✓			✓	✓	✓
Прошедшее время	✓	✓				✓	✓	✓
Состояние потока	✓		✓		✓	✓	✓	✓
Причина перехода в состояние ожидания	✓		✓		✓	✓	✓	✓
Последняя ошибка					✓		✓	
% загруженности процессора	✓			✓			✓	
% работы в пользовательском режиме	✓			✓			✓	
% работы в привилегированном режиме	✓			✓			✓	
Адрес TEB						✓		
Адрес ETHREAD						✓		
Объекты, ожидаемые данным потоком						✓		

Process Explorer позволяет наблюдать за активностью потоков в процессе. Это особенно важно, когда вы пытаетесь понять, почему процесс зависает или запускается какой-то процесс, служащий хостом для множества сервисов (например, Svchost.exe, Dllhost.exe, Inetinfo.exe или System).

Для просмотра потоков в процессе выберите этот процесс и откройте окно его свойств двойным щелчком. Потом перейдите на вкладку Threads. На этой вкладке отображается список потоков в процессе. Для каждого потока показывается процентная доля использованного процессорного времени (с

учетом заданного интервала обновления), число переключений контекста для данного потока и его стартовый адрес. Поддерживается сортировка по любому из этих трех столбцов.

Новые потоки выделяются зеленым, а существующие — красным. (Длительность подсветки настраивается в Options.) Это помогает обнаруживать создание лишних потоков в процессе. (Как правило, потоки должны создаваться при запуске процесса, а не при каждой обработке какого-либо запроса внутри процесса.)

Когда вы поочередно выбираете потоки в списке, Process Explorer отображает их идентификаторы, время запуска, состояние, счетчики использования процессорного времени, число переключений контекстов, а также базовый и текущий приоритеты. Кнопка Kill позволяет принудительно завершать индивидуальные потоки, но пользоваться ею следует с крайней осторожностью.

Разница в числе переключений контекста (context switch delta) отражает, сколько раз потоки начинали работать в течение периода обновления, указанного в Process Explorer. Это еще один способ определения активности потоков. В некоторых отношениях он даже лучше, так как многие потоки выполняются в течение лишь очень короткого времени и поэтому крайне редко попадают в список текущих потоков. Например, если вы добавите столбец с разницей в числе переключений контекстов к тому, что показывается для процесса и отсортируете по этому столбцу, то увидите процессы, в которых потоки выполняются, но используют очень мало процессорного времени или вообще его не используют.

Стартовый адрес потока выводится в виде «*module*!*function*», где *module* — имя EXE или DLL. Имя функции извлекается из файла символов для данного модуля (см. эксперимент «Просмотр детальных сведений о процессах с помощью Process Explorer» в главе 1). Если вы точно не знаете, что это за модуль, нажмите кнопку Module, и появится окно свойств модуля, где содержится данная функция.

ПРИМЕЧАНИЕ Для потоков, созданных Windows-функцией *CreateThread*, Process Explorer показывает функцию, переданную в *CreateThread*, а не истинную стартовую функцию потока. Это связано с тем, что все Windows-потоки запускаются в общей стартовой функции-оболочке для процессов или потоков (*BaseProcessStart* либо *BaseThreadStart* в Kernel32.dll). Если бы Process Explorer выводил истинный стартовый адрес, то казалось бы, что большинство потоков в процессе были запущены по одному адресу, а это вряд ли помогло бы понять, какой код выполняется потоком.

Однако одного стартового адреса потока может оказаться недостаточно для того, чтобы выяснить, что именно делает поток и какой компонент внутри процесса отвечает за использование процессорного времени этим потоком. Это особенно верно, если стартовый адрес потока относится к универсальной стартовой функции (например, если имя функции не указывает на

то, что делает данный поток). Тогда может помочь изучение стека потока. Для его просмотра дважды щелкните интересующий вас поток (или выберите этот поток и нажмите кнопку Stack). Process Explorer покажет стек потока (пользовательского режима и режима ядра, если поток был в последнем режиме).

ПРИМЕЧАНИЕ Отладчики пользовательского режима (Windbg, Ntsd и Cdb) тоже позволяют подключаться к процессу и просматривать стек потока, но Process Explorer выводит стек как пользовательского режима, так и режима ядра простым нажатием одной кнопки. Стеки пользовательского режима и режима ядра можно просмотреть и с помощью Livekd от *www.sysinternals.com*, но эта утилита гораздо сложнее в использовании. Кстати, при работе Windbg в режиме локальной отладки ядра, поддерживаемом только в Windows XP и Windows Server 2003, увидеть содержимое стеков потоков нельзя.

Просмотр стека потока полезен и при поиске причины зависания процесса. Например, на одной системе Microsoft PowerPoint зависал при запуске на минуту. Чтобы понять причину этого зависания, с помощью Process Explorer изучили стек одного из потоков в процессе. Результат приведен на рис. 6-10.

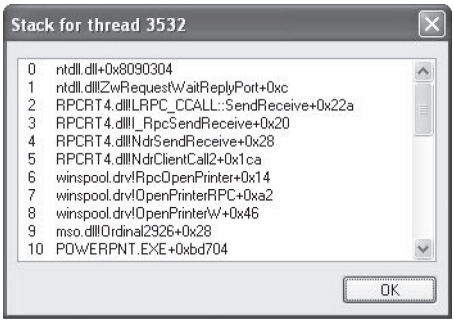


Рис. 6-10. *Стек зависшего потока в PowerPoint*

Как видите, PowerPoint (строка 10) вызвал функцию в Mso.dll (основной Microsoft Office DLL), которая обратилась к функции *OpenPrinterW* в Winspool.drv (DLL, используемой для подключения к принтерам). Затем Winspool.drv пересылает запрос функции *OpenPrinterRPC*, а та вызывает функцию в DLL исполняющей среды RPC, сообщая, что запрос посылается удаленному принтеру. Вот так, не зная деталей внутреннего устройства PowerPoint, по именам модулей и функций в стеке потока можно понять, что поток ждет соединения с сетевым принтером. В данной системе был сетевой принтер, который не отвечал, что и объясняет задержку в запуске PowerPoint. (Приложения Microsoft Office соединяются со всеми сконфигурированными принтерами при запуске.) Соединение с тем принтером было удалено из пользовательской системы, и проблема исчезла.

Планирование потоков

Здесь описываются стратегии и алгоритмы планирования в Windows. В первом разделе этой части материалов рассматриваются принципы планирования в Windows и даются определения ключевых терминов. Уровни приоритета обсуждаются с точки зрения как Windows API, так и ядра. После обзора сопутствующих Windows-функций и утилит подробно анализируются — сначала в однопроцессорных системах, а затем и в многопроцессорных — алгоритмы и структуры данных, используемые подсистемой планирования Windows.

Обзор планирования в Windows

В Windows реализована подсистема вытесняющего планирования на основе уровней приоритета, в которой всегда выполняется поток с наибольшим приоритетом, готовый к выполнению. Однако выбор потока для выполнения может быть ограничен набором процессоров, на которых он может работать. Это явление называется *привязкой к процессорам* (processor affinity). По умолчанию поток выполняется на любом доступном процессоре, но вы можете изменить привязку к процессорам через Windows-функции планирования, перечисленные в таблице 6-14 (см. далее в этой главе), или заданием маски привязки в заголовке образа.

ЭКСПЕРИМЕНТ: просмотр потоков, готовых к выполнению

Список потоков, готовых к выполнению, можно увидеть с помощью команды *!ready* отладчика ядра. Она выводит поток или список потоков, готовых к выполнению (на каждом уровне приоритета отдельно). В следующем примере к выполнению готовы два потока с приоритетом 10 и шесть потоков — с приоритетом 8. Поскольку эта информация получена в однопроцессорной системе с использованием LiveKd, текущим потоком всегда является отладчик ядра (Kd или WinDbg).

```
kd> !ready 1
Ready Threads at priority 10
 THREAD 810de030  Cid 490.4a8 Teb: 7ffd9000  Win32Thread: e297e008 READY
 THREAD 81110030  Cid 490.48c Teb: 7ffde000  Win32Thread: e29425a8 READY
Ready Threads at priority 8
 THREAD 811fe790  Cid 23c.274 Teb: 7ffdb000  Win32Thread: e258cda8 READY
 THREAD 810bec70  Cid 23c.50c Teb: 7ffd9000  Win32Thread: e2ccf748 READY
 THREAD 8003a950  Cid 23c.550 Teb: 7ffda000  Win32Thread: e29a7ae8 READY
 THREAD 85ac2db0  Cid 23c.5e4 Teb: 7ffd8000  Win32Thread: e297a9e8 READY
 THREAD 827318d0  Cid 514.560 Teb: 7ffd9000  Win32Thread: 00000000 READY
 THREAD 8117adb0  Cid 2d4.338 Teb: 7ffaf000  Win32Thread: 00000000 READY
```

Выбранный для выполнения поток работает в течение некоего периода, называемого *квантом*. Квант определяет, сколько времени будет выполнять-

ся поток, пока не наступит очередь другого потока с тем же приоритетом (или более высоким, что возможно в многопроцессорной системе). Длительность квантов зависит от трех факторов: конфигурационных параметров системы (длинные или короткие кванты), статуса процесса (активный или фоновый) и использования объекта «задание» для изменения длительности квантов. (Подробнее о квантах см. раздел «Квант» далее в этой главе.) Однако поток может не полностью использовать свой квант. Поскольку в Windows реализован вытесняющий планировщик, то происходит вот что. Как только другой поток с более высоким приоритетом готов к выполнению, текущий поток вытесняется, даже если его квант еще не истек. Фактически поток может быть выбран следующим для выполнения и вытеснен, не успев воспользоваться своим квантом!

Код Windows, отвечающий за планирование, реализован в ядре. Поскольку этот код рассредоточен по ядру, единого модуля или процедуры с названием «планировщик» нет. Совокупность процедур, выполняющих эти обязанности, называется *диспетчером ядра* (kernel's dispatcher). Диспетчеризация потоков может быть вызвана любым из следующих событий.

- Поток готов к выполнению — например, он только что создан или вышел из состояния ожидания.
- Поток выходит из состояния Running (выполняется), так как его квант истек или поток завершается либо переходит в состояние ожидания.
- Приоритет потока изменяется в результате вызова системного сервиса или самой Windows.
- Изменяется привязка к процессорам, из-за чего поток больше не может работать на процессоре, на котором он выполнялся.

В любом случае Windows должна определить, какой поток выполнять следующим. Выбрав новый поток, Windows *переключает контекст*. Эта операция заключается в сохранении параметров состояния машины, связанных с выполняемым потоком, и загрузке аналогичных параметров для другого потока, после чего начинается выполнение нового потока.

Как уже говорилось, планирование в Windows осуществляется на уровне потоков. Этот подход станет понятен, если вы вспомните, что сами процессы не выполняются, а лишь предоставляют ресурсы и контекст для выполнения потоков. Поскольку решения, принимаемые в ходе планирования, касаются исключительно потоков, система не обращает внимания на то, какому процессу принадлежит тот или иной поток. Так, если у процесса A есть 10, у процесса B — 2 готовых к выполнению потока, и все 12 имеют одинаковый приоритет, каждый из потоков теоретически получит 1/12 процессорного времени, потому что Windows не станет поровну делить процессорное время между двумя процессами.

Чтобы понять алгоритмы планирования потоков, вы должны сначала разобраться в уровнях приоритета, используемых Windows.

Уровни приоритета

Как показано на рис. 6-11, в Windows предусмотрено 32 уровня приоритета — от 0 до 31. Эти значения группируются так:

- шестнадцать уровней реального времени (16–31);
- пятнадцать варьируемых (динамических) уровней (1–15);
- один системный уровень (0), зарезервированный для потока обнуления страниц (zero page thread).

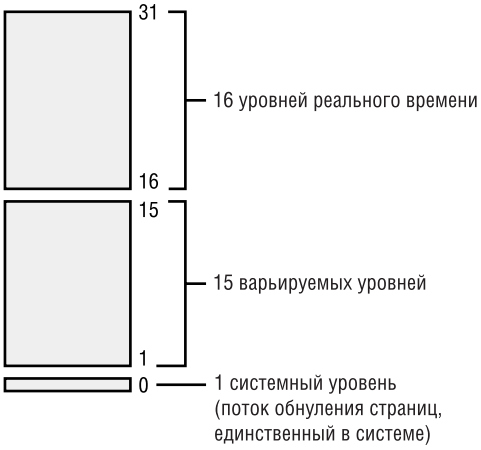


Рис. 6-11. *Уровни приоритета потоков*

Уровни приоритета потока назначаются с учетом двух разных точек зрения — Windows API и ядра Windows. Windows API сначала упорядочивает процессы по классам приоритета, назначенным при их создании [Real-time (реального времени), High (высокий), Above Normal (выше обычного), Normal (обычный), Below Normal (ниже обычного) и Idle (простаивающий)], а затем — по относительному приоритету индивидуальных потоков в рамках этих процессов [Time-critical (критичный по времени), Highest (наивысший), Above-normal (выше обычного), Normal (обычный), Below-normal (ниже обычного), Lowest (наименьший) и Idle (простаивающий)].

Базовый приоритет каждого потока в Windows API устанавливается, исходя из класса приоритета его процесса и относительного приоритета самого потока. Связь между приоритетами Windows API и внутренними приоритетами ядра Windows (в числовой форме) показана на рис. 6-12.

Если у процесса только одно значение приоритета (базовое), то у каждого потока их два: текущее и базовое. Решения, связанные с планированием, принимаются на основе текущего приоритета. Как поясняется в следующем разделе, в определенных обстоятельствах система может на короткое время повышать приоритеты потоков в динамическом диапазоне (1–15). Windows никогда не изменяет приоритеты потоков в диапазоне реального времени (16–31), поэтому у таких потоков базовый приоритет идентичен текущему.

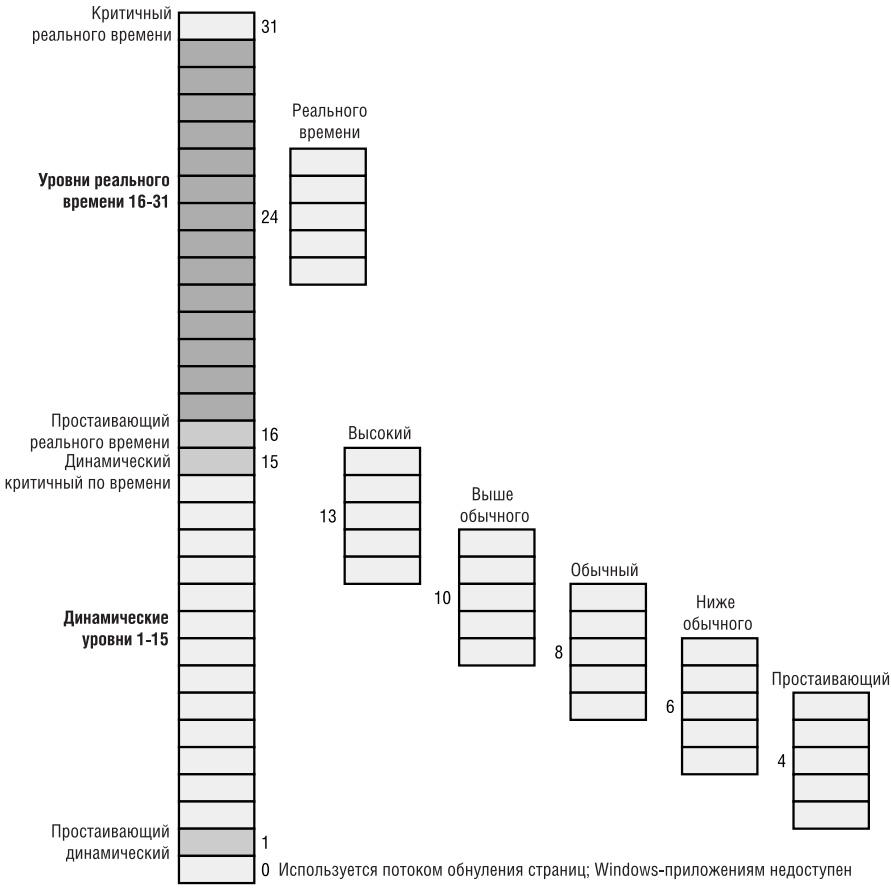


Рис. 6-12. *Взаимосвязь приоритетов в ядре и Windows API*

Начальный базовый приоритет потока наследуется от базового приоритета процесса, а тот наследует его от родительского процесса. Это поведение можно изменить при вызове Windows-функции *CreateProcess* или команды START. Приоритет процесса можно изменить и после его создания, используя функцию *SetPriorityClass* или различные утилиты, предоставляющие доступ к этой функции через UI, например диспетчер задач и Process Explorer. В частности, вы можете понизить приоритет процесса, интенсивно использующего процессорное время, чтобы он не мешал обычным операциям в системе. Смена приоритета процесса влечет за собой смену приоритетов всех его потоков, но их относительные приоритеты остаются прежними. Но изменение приоритетов индивидуальных потоков внутри процесса обычно не имеет смысла, потому что вы не знаете, чем именно занимается каждый из них (если только сами не пишете программу или не располагаете исходным кодом); так что изменение относительных приоритетов потоков может привести к неадекватному поведению этого приложения.

Обычно базовый приоритет процесса (а значит, и базовый приоритет первичного потока) по умолчанию равен значению из середины диапазонов при-

оритетов процессов (24, 13, 10, 8, 6 или 4). Однако базовый приоритет некоторых системных процессов (например, диспетчера сеансов, контроллера сервисов и сервера локальной аутентификации) несколько превышает значение по умолчанию для класса Normal (8). Более высокий базовый приоритет по умолчанию обеспечивает запуск потоков этих процессов с приоритетом выше 8. Чтобы изменить свой начальный базовый приоритет, такие системные процессы используют внутреннюю функцию *NtSetInformationProcess*.

Функции Windows API, связанные с планированием

Эти функции перечислены в таблице 6-14 (более подробную информацию см. в справочной документации Windows API).

Таблица 6-14. *API-функции планирования и их назначение*

API-функция	Описание
SuspendThread *ResumeThread*	Приостанавливает/возобновляет поток
GetPriorityClass *SetPriorityClass*	Возвращает/устанавливает класс приоритета процесса (базовый приоритет)
GetThreadPriority *SetThreadPriority*	Возвращает/устанавливает приоритет потока (относительный базовому приоритету его процесса)
GetProcessAffinityMask *SetProcessAffinityMask*	Возвращает/устанавливает маску привязки процесса к процессорам
SetThreadAffinityMask	Устанавливает маску привязки потока (которая должна быть подмножеством маски привязки процесса) к конкретному набору процессоров, ограничивая доступные для выполнения этого потока процессоры
SetInformationJobObject	Задает атрибуты для задания; некоторые из них влияют на планирование, изменяя, например, привязку к процессорам и приоритет (описание объекта «задание» см. в разделе «Объекты-задания» далее в этой главе)
GetLogicalProcessor-Information	Возвращает детальные сведения о конфигурации процессоров [для систем с поддержкой логических процессоров (hyperthreaded systems) и NUMA]
GetThreadPriorityBoost *SetThreadPriorityBoost*	Возвращает информацию о динамическом повышении приоритета потока или разрешает такое повышение приоритета (только для потоков с приоритетами из динамического диапазона)
SetThreadIdealProcessor	Задает предпочтительный процессор для данного потока, но не ограничивает им набор процессоров, доступных этому потоку
GetProcessPriorityBoost *SetProcessPriorityBoost*	Возвращает/устанавливает статус динамического повышения приоритета данного процесса по умолчанию (используется для установки этого статуса при создании потока)
SwitchToThread	Переключает процессор на выполнение другого потока (с приоритетом от 1 и выше), готового к выполнению на текущем процессоре

Таблица 6-14. (*окончание*)

API-функция	Описание
Sleep	Переводит текущий поток в состояние ожидания на указанное время (в мс); нулевое значение отбирает у потока остаток его кванта
SleepEx	Переводит текущий поток в состояние ожидания до тех пор, пока не закончится операция ввода-вывода, пока не истечет указанный временной интервал или пока в очереди потока не появится APC

Сопутствующие утилиты

В следующей таблице перечислены утилиты, сообщающие информацию о планировании потоков. Базовый приоритет процесса можно увидеть (и изменить) с помощью диспетчера задач, Process Explorer, Pview или Pviewer. Заметьте, что Process Explorer позволяет уничтожать отдельные потоки в любых процессах. Но, конечно же, этой возможностью следует пользоваться с крайней осторожностью.

Приоритеты потоков можно просмотреть в оснастке Performance (Производительность), а также с помощью утилит Process Explorer, Plist, Pview, Pviewer и Pstat. Хотя повышение или понижение приоритета процесса может оказаться весьма полезным, изменение приоритетов индивидуальных потоков внутри процесса, как правило, не имеет смысла, потому что постороннему человеку не известно, что делают эти потоки и почему важны именно такие их относительные приоритеты.

Объект	Taskman	Perfmon	Pviewer	Pview	Pstat	KD *!thread*	Process Explorer
Класс приоритета процесса	✓		✓	✓			✓
Базовый приоритет процесса		✓			✓		✓
Базовый приоритет потока		✓					✓
Текущий приоритет потока		✓	✓	✓	✓	✓	✓

Единственный способ задать начальный класс приоритета для процесса — использовать команду *start* в командной строке Windows. Если вы хотите, чтобы некая программа каждый раз запускалась с определенным приоритетом, то можете создать для нее ярлык и указать команду запуска, предварив ее **cmd /c**. Это приведет к появлению окна командной строки, выполнению команды и последующему закрытию этого окна. Например, чтобы запустить Notepad в процессе с низким приоритетом, в свойствах ярлыка должна быть задана команда **cmd /c start /low notepad.exe**.

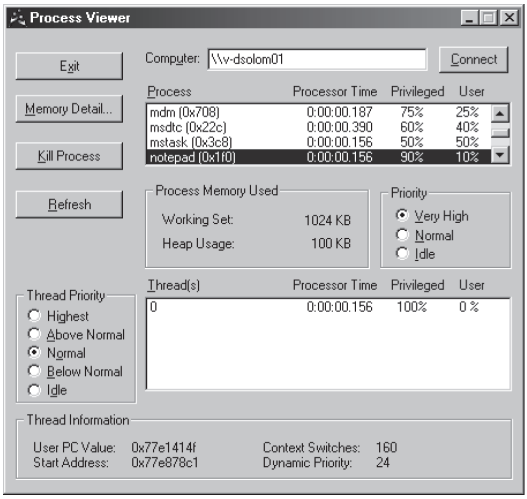
ЭКСПЕРИМЕНТ: исследуем и задаем приоритеты процессов и потоков

Попробуйте провести такой эксперимент.

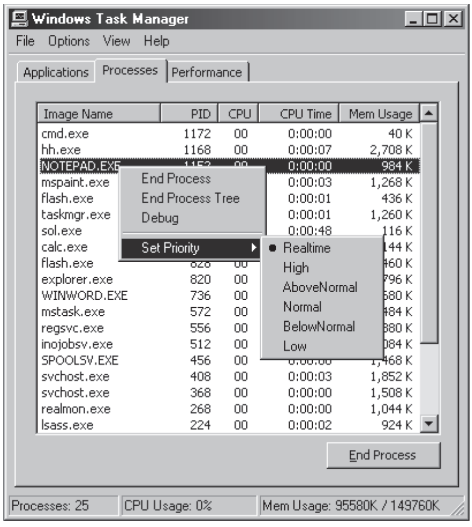
1. Наберите в командной строке **start /realtime notepad**. На экране появится окно Notepad.

см. след. стр.

2. Запустите утилиту Process Explorer или Process Viewer (Pviewer.exe) из Support Tools и выберите Notepad.exe из списка процессов, как показано ниже. Заметьте, что динамический приоритет потока Notepad равен 24. Это значение совпадает со значением приоритета реального времени на рис. 6-12.



3. Аналогичную информацию можно получить в диспетчере задач. Для его запуска нажмите клавиши Ctrl+Shift+Esc и перейдите на вкладку Processes (Процессы). Щелкните правой кнопкой мыши процесс Notepad.exe и выберите команду Set Priority (Приоритет). Вы увидите, что класс приоритета потока относится к Realtime (Реального времени), как показано на следующей иллюстрации.



Диспетчер системных ресурсов Windows

В Windows Server 2003 Enterprise Edition и Windows Server 2003 Datacenter Edition включен необязательный компонент, который называется диспетчером системных ресурсов Windows (Windows System Resource Manager, WSRM). Он позволяет администратору настраивать правила политики, указывающие для процессов использование процессорного времени, параметры привязки к процессорам и лимиты на физическую и виртуальную память. Кроме того, WSRM может генерировать отчеты по использованию ресурсов, удобные для учета и проверки уровня обслуживания по договорам с пользователями.

Такие правила могут быть применены к конкретным приложениям, пользователям или группам и действовать в определенные периоды или постоянно.

После того как вы сформировали политику выделения ресурсов для управления определенными процессами, служба WSRM будет вести мониторинг потребления ими процессорного времени и регулировать их базовые приоритеты, если эти процессы будут использовать процессорного времени больше или меньше, чем было установлено вами.

Ограничение физической памяти достигается заданием максимального размера рабочего набора через функцию *SetProcessWorkingSetSizeEx*, а ограничение виртуальной памяти реализуется самой службой (о лимитах на объемы физической и виртуальной памяти см. главу 7). Если заданный лимит превышен, WSRM — в зависимости от настроек — может уничтожать процессы или создавать соответствующую запись в журнале событий. Последнее позволяет выявить процесс с утечкой памяти до того, как он займет всю переданную виртуальную память в системе. Заметьте, что лимиты на память, установленные в WSRM, не применяются к памяти Address Windowing Extensions (AWE), памяти больших страниц (large page memory) или памяти ядра (пулу подкачиваемых или неподкачиваемых страниц).

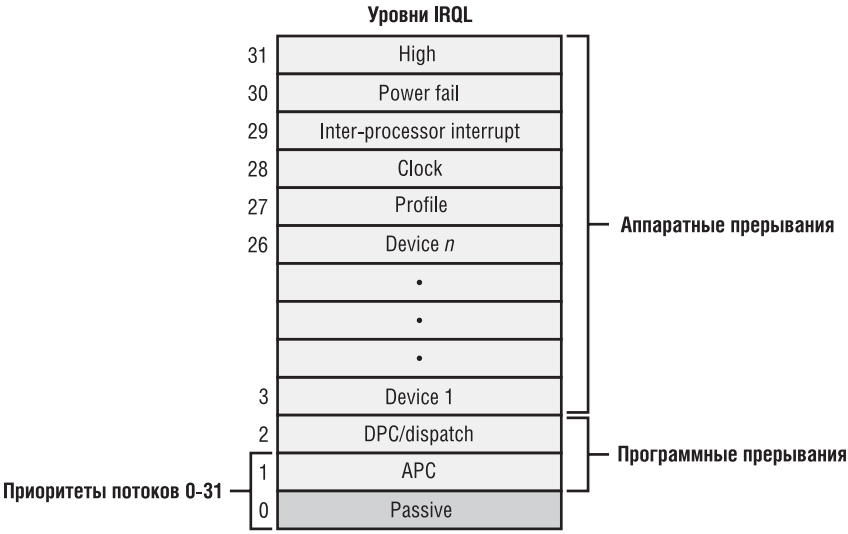
Приоритеты реального времени

Вы можете повысить или понизить приоритет потока любого приложения в динамическом диапазоне; однако, чтобы задать значение из диапазона реального времени, у вас должна быть привилегия Increase Scheduling Priority. Учтите, что многие важные системные потоки режима ядра выполняются в диапазоне приоритетов реального времени. Поэтому, если потоки слишком долго выполняются с приоритетом этого диапазона, они могут блокировать критичные системные функции (например в диспетчере памяти, диспетчере кэша или драйверах устройств).

ПРИМЕЧАНИЕ Как показано на следующей иллюстрации, где изображены уровни запросов прерываний (Interrupt Request Levels, IRQL) на платформе x86, в Windows имеется набор приоритетов, называемых приоритетами реального времени, но они не являются таковыми в общепринятом смысле этого термина, так как Windows не относится к операционным системам реального времени. Подробнее на эту тему см. врезку «Windows и обработка данных в реальном времени» в главе 3, а также статью «Real-Time Systems and Microsoft Windows NT» в MSDN Library.

Уровни прерываний и уровни приоритета

Как показано на следующей иллюстрации, потоки обычно выполняются при IRQL, равном 0 или 1. (Описание уровней прерываний в Windows см. в главе 3.) Потоки пользовательского режима всегда выполняются при IRQL, равном 0. Ввиду этого ни один поток пользовательского режима независимо от его приоритета не в состоянии блокировать аппаратные прерывания (хотя потоки с высоким приоритетом из диапазона реального времени способны блокировать важные системные потоки). При IRQL, равном 1, работают только APC режима ядра, поскольку они прерывают выполнение потоков (об APC см. главу 3). Кроме того, потоки, выполняемые в режиме ядра, могут повышать IRQL, например при обработке системного вызова, требующего диспетчеризации потоков.



Состояния потоков

Прежде чем перейти к алгоритмам планирования потоков, вы должны разобраться, в каких состояниях могут находиться потоки в процессе выполнения

в Windows 2000 и Windows XP. Соответствующая схема дана на рис. 6-13 [числовые значения отражают показатели счетчика производительности Thread: thread state (Поток: Состояние потока)].

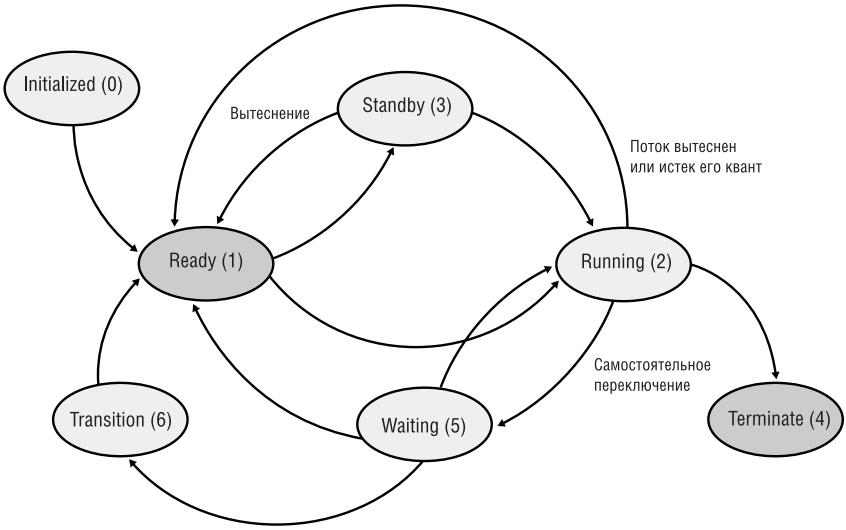


Рис. 6-13. *Состояния потоков в Windows 2000 и Windows XP*

Вот что представляют собой состояния потока.

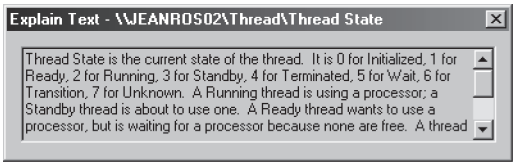
- **Ready (готов)** Поток в состоянии готовности ожидает выполнения. Выбирая следующий поток для выполнения, диспетчер принимает во внимание только пул потоков, готовых к выполнению.
- **Standby (простаивает)** Поток в этом состоянии уже выбран следующим для выполнения на конкретном процессоре. В подходящий момент диспетчер переключает контекст на этот поток. В состоянии Standby может находиться только один поток для каждого процессора в системе. Заметьте, что поток может быть вытеснен даже в этом состоянии (если, например, до начала выполнения потока, который пока находится в состоянии Standby, к выполнению будет готов поток с более высоким приоритетом).
- **Running (выполняется)** Поток переходит в это состояние и начинает выполняться сразу после того, как диспетчер переключает на него контекст. Выполнение потока прекращается, как только он завершается, вытесняется потоком с более высоким приоритетом, переключает контекст на другой поток, самостоятельно переходит в состояние ожидания или истекает выделенный ему квант процессорного времени (и другой поток с тем же приоритетом готов к выполнению).
- **Waiting (ожидает)** Поток входит в состояние Waiting несколькими способами. Он может самостоятельно начать ожидание на синхронизирующем объекте или его вынуждает к этому подсистема окружения. По окончании ожидания поток — в зависимости от приоритета — либо немедленно начинает выполняться, либо переходит в состояние Ready.

- **Transition (переходное состояние)** Поток переходит в это состояние, если он готов к выполнению, но его стек ядра выгружен из памяти. Как только этот стек загружается в память, поток переходит в состояние Ready.
- **Terminated (завершен)** Заканчивая выполнение, поток переходит в состояние Terminated. После этого блок потока исполнительной системы (структура данных в пуле неподкачиваемой памяти, описывающая данный поток) может быть удален, а может быть и не удален — это уже определяется диспетчером объектов.
- **Initialized (инициализирован)** В это состояние поток входит в процессе своего создания.

ЭКСПЕРИМЕНТ: изменение состояний потоков при планировании

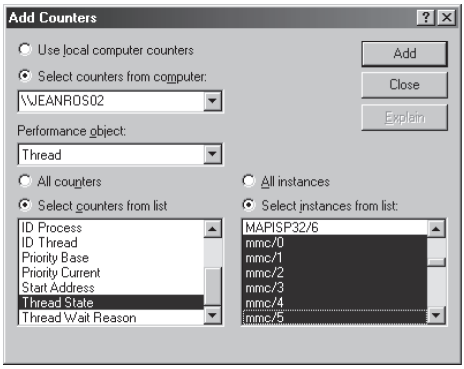
Вы можете понаблюдать за изменением этих состояний с помощью оснастки Performance. Она может оказаться полезной в отладке многопоточных приложений, если вам нужно проверить состояние потоков, выполняемых в вашем процессе.

1. Запустите стандартную программу Notepad (Блокнот) (Notepad.exe).
2. Запустите оснастку Performance (Производительность), открыв в меню Start (Пуск) подменю Programs (Программы) и Administrative Tools (Администрирование), а затем выбрав команду Performance (Производительность).
3. Выберите режим просмотра диаграмм (если установлен какой-то другой).
4. Щелкните график правой кнопкой мыши и выберите команду Properties (Свойства).
5. Откройте вкладку Graph (График) и установите максимальное значение вертикальной шкалы равным 7. (Состояниям потоков соответствуют числа от 0 до 7). Щелкните кнопку OK.
6. Щелкните на панели инструментов кнопку Add (Добавить), чтобы открыть диалоговое окно Add Counters (Добавить счетчики).
7. Выберите в списке объект Thread (Поток), а затем — счетчик Thread State (Состояние потока). Определение его значений вы увидите, щелкнув кнопку Explain (Объяснение), как показано ниже.

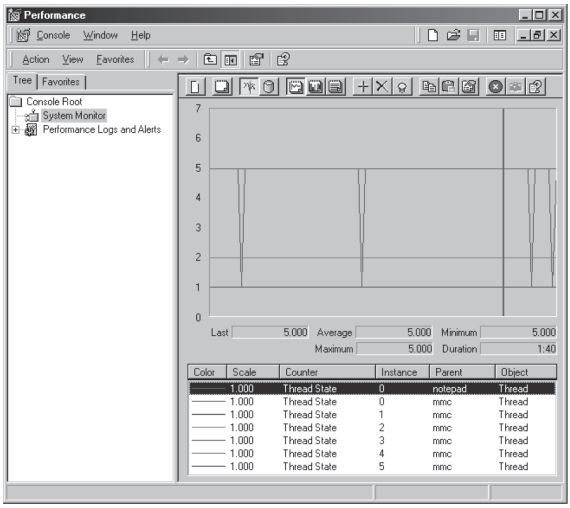


8. Прокрутите список вхождений до строки notepad/0 (это процесс Notepad), выделите его и щелкните кнопку Add (Добавить).
9. Прокрутите список назад до процесса Mmc (это процесс Microsoft Management Console, в котором выполняется ActiveX-элемент System Monitor), выберите все его потоки (mmc/0, mmc/1 и т. д.) и до-

бавьте их на график, щелкнув кнопку Add. Прежде чем щелкнуть кнопку Add, вы должны увидеть диалоговое окно, аналогичное показанному ниже.



10. Теперь закройте диалоговое окно Add Counters, щелкнув кнопку Close (Закрыть).
11. Вы должны увидеть, что поток Notepad (верхняя линия графика) находится в состоянии 5. Как вы уже знаете, значение 5 соответствует состоянию Waiting. (В данном случае поток ждет GUI-ввода.)



12. Заметьте, что один из потоков процесса Mmc (выполняющий оснастку Performance) находится в состоянии Running (значение 2). Этот поток всегда выполняется, так как постоянно запрашивает состояние других потоков.
13. Вы никогда не увидите процесс Notepad в состоянии Running (если только не используете многопроцессорную систему), поскольку в этом состоянии всегда находится Mmc, собирая данные о состоянии отслеживаемых потоков.

Схема состояний потоков в Windows Server 2003 показана на рис. 6-14. Обратите внимание на новое состояние Deferred Ready (готов, отложен). Это состояние используется для потоков, выбранных для выполнения на конкретном процессоре, но пока не запланированных к выполнению. Это новое состояние предназначено для того, чтобы ядро могло свести к минимуму срок применения общесистемной блокировки к базе данных планирования (scheduling database). (Этот процесс подробно описывается в разделе «База данных диспетчера ядра в многопроцессорной системе».)

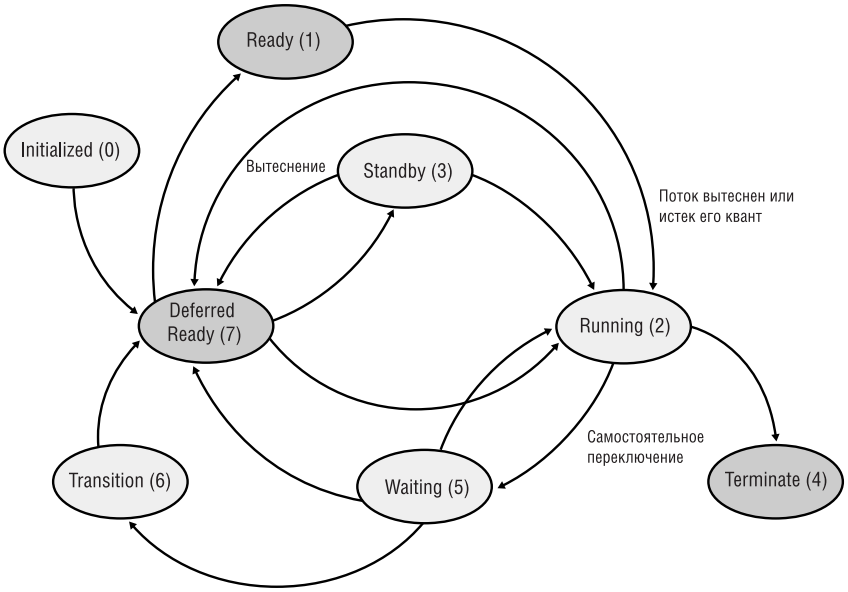


Рис. 6-14. *Состояния потоков в Windows Server 2003*

База данных диспетчера ядра

Для принятия решений при планировании потоков ядро поддерживает набор структур данных, в совокупности известных как *база данных диспетчера ядра* (dispatcher database) (рис. 6-15). Эта база данных позволяет отслеживать потоки, ждущие выполнения, и потоки, выполняемые на тех или иных процессорах.

ПРИМЕЧАНИЕ База данных диспетчера ядра в однопроцессорной системе имеет ту же структуру, что и в многопроцессорных системах Windows 2000 и Windows XP, но отличается от структуры такой базы данных в системах Windows Server 2003. Эти различия, а также иной алгоритм выбора потоков для выполнения в многопроцессорных системах поясняются в разделе «Многопроцессорные системы».

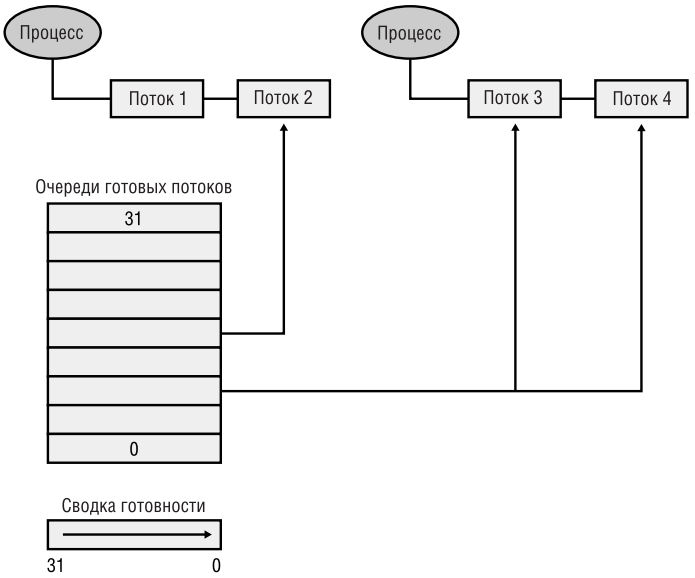


Рис. 6-15. *База данных диспетчера ядра (однопроцессорная и многопроцессорная в Windows 2000/XP)*

Очереди готовых потоков (ready queues) диспетчера ядра включают потоки в состоянии Ready, ожидающие выделения им процессорного времени. Для каждого из 32 уровней приоритета существует по одной очереди. Для ускорения выбора потока, подлежащего выполнению или вытеснению, Windows поддерживает 32-битную маску, называемую *сводкой готовности* (ready summary) (*KiReadySummary*). Каждый установленный в ней бит указывает на присутствие одного или более потоков в очереди готовых потоков для данного уровня приоритета (бит 0 соответствует приоритету 0, бит 1 — приоритету 1 и т. д.).

В таблице 6-15 перечислены переменные ядра, связанные с планированием потоков в однопроцессорных системах.

Таблица 6-15. *Переменные ядра, связанные с планированием потоков*

Переменная	Тип	Описание
KiReadySummary	32-битная маска	Битовая маска уровней приоритета, имеющихся у одного или нескольких готовых потоков
KiDispatcherReady-ListHead	Массив из 32 записей	Список заголовков для 32 очередей готовых потоков

В однопроцессорных системах база данных диспетчера ядра синхронизируется повышением IRQL до уровня «DPC/dispatch» и SYNCH_LEVEL (оба определены как уровень 2). (Об уровнях приоритета прерываний см. раздел «Диспетчеризация ловушек» главы 3.) Такое повышение IRQL не дает другим потокам прервать диспетчеризацию потоков, так как потоки обычно выпол-

няются при IRQL 0 или 1. В многопроцессорных системах одного повышения IRQL мало, потому что каждый процессор может одновременно увеличить IRQL до одного уровня и попытаться обратиться к базе данных диспетчера ядра. Как Windows синхронизирует доступ к этой базе данных в многопроцессорных системах, поясняется в разделе «Многопроцессорные системы» далее в этой главе.

Квант

Как уже говорилось, квант — это интервал процессорного времени, отведенный потоку для выполнения. По его окончании Windows проверяет, ожидает ли выполнения другой поток с таким же уровнем приоритета. Если на момент истечения кванта других потоков с тем же уровнем приоритета нет, Windows выделяет текущему потоку еще один квант.

По умолчанию в Windows 2000 Professional и Windows XP потоки выполняются в течение 2 интервалов таймера (clock intervals), а в системах Windows Server — 12. (Как изменить эти значения, мы объясним позже.) В серверных системах величина кванта увеличена для того, чтобы свести к минимуму переключение контекста. Получая больший квант, серверные приложения, которые пробуждаются при получении клиентского запроса, имеют больше шансов выполнить запрос и вернуться в состояние ожидания до истечения выделенного кванта.

Длительность интервала таймера зависит от аппаратной платформы и определяется HAL, а не ядром. Например, этот интервал на большинстве однопроцессорных x86-систем составляет 10 мс, а на большинстве многопроцессорных x86-систем — около 15 мс. (Как проверить реальный интервал системного таймера, см. в следующем эксперименте.)

ЭКСПЕРИМЕНТ: определяем величину интервала системного таймера

Windows-функция *GetSystemTimeAdjustment* возвращает величину интервала системного таймера. Для ее определения запустите программу Clockres с *www.sysinternals.com*. Вот что это программа выводит на однопроцессорной x86-системе:

```
C:\>clockres

ClockRes - View the system clock resolution
By Mark Russinovich
SysInternals - www.sysinternals.com

The system clock interval is 10.014400 ms
```

Учет квантов времени

Величина кванта для каждого процесса хранится в блоке процесса ядра. Это значение используется, когда потоку предоставляется новый квант. Когда поток выполняется, его квант уменьшается по истечении каждого интерва-

ла таймера, и в конечном счете срабатывает алгоритм обработки завершения кванта. Если имеется другой поток с тем же приоритетом, ожидающий выполнения, происходит переключение контекста на следующий поток в очереди готовых потоков. Заметьте: когда системный таймер прерывает DPC или процедуру обработки другого прерывания, квант выполнявшегося потока все равно уменьшается, даже если этот поток не успел отработать полный интервал таймера. Если бы это было не так и если бы аппаратное прерывание или DPC появилось непосредственно перед прерыванием таймера, квант потока мог бы вообще никогда не уменьшиться.

Внутренне величина кванта хранится как число тактов таймера, умноженное на 3. То есть в Windows 2000 и Windows XP потоки по умолчанию получают квант величиной 6 (2 * 3), в Windows Server — 36 (12 * 3). Всякий раз, когда возникает прерывание таймера, процедура его обработки вычитает из кванта потока постоянную величину (3).

Почему квант внутренне хранится как величина, кратная трем квантовым единицам за один такт системного таймера? Это сделано для того, чтобы можно было уменьшать значение кванта по завершении ожидания. Когда поток с текущим приоритетом ниже 16 и базовым приоритетом ниже 14 запускает функцию ожидания (*WaitForSingleObject* или *WaitForMultipleObjects*) и его запрос на доступ удовлетворяется немедленно (например, он не переходит в состояние ожидания), его квант уменьшается на одну единицу. Благодаря этому кванты ожидающих потоков в конечном счете заканчиваются.

Если запрос на доступ не удовлетворяется немедленно, кванты потоков с приоритетом ниже 16 тоже уменьшаются на одну единицу (кроме случая, когда поток пробуждается для выполнения APC ядра). Но перед такой операцией квант потока с приоритетом 14 или выше сбрасывается. Это делается и для потоков с приоритетом менее 14, если они не выполняются при специально повышенном приоритете (как, например, в случае фоновых процессов или при недостаточном выделении процессорного времени) и если их приоритет повышен в результате выхода из состояния ожидания (unwait operation). (Динамическое повышение приоритета поясняется в следующем разделе.)

Управление величиной кванта

Вы можете изменить квант для потоков всех процессов, но выбор ограничен всего двумя значениями: короткий квант (2 такта таймера, используется по умолчанию для клиентских компьютеров) или длинный (12 тактов таймера, используется по умолчанию для серверных систем).

> **ПРИМЕЧАНИЕ** Используя объект «задание» в системе с длинными квантами, вы можете указать другие величины квантов для процессов в задании. Более подробную информацию об объектах «задание» см. в разделе «Объекты-задания» далее в этой главе.

Для изменения величины кванта в Windows 2000 щелкните правой кнопкой мыши My Computer (Мой компьютер), выберите Properties (Свойства),

перейдите на вкладку Advanced (Дополнительно), а затем щелкните кнопку Performance Options (Параметры быстродействия). Вы увидите диалоговое окно, показанное на рис. 6-16.

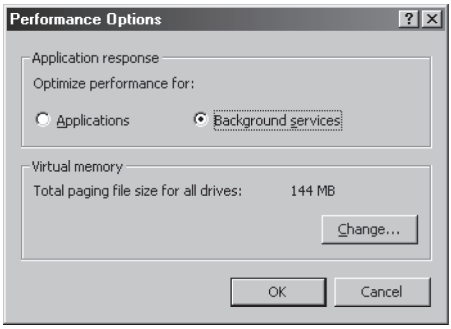


Рис. 6-16. *Задание кванта в Windows 2000*

Для изменения величины кванта в Windows XP или Windows Server 2003 щелкните правой кнопкой мыши My Computer (Мой компьютер), выберите Properties (Свойства), перейдите на вкладку Advanced (Дополнительно), щелкните кнопку Settings (Параметры) в разделе Performance (Быстродействие), а затем перейдите на вкладку Advanced (Дополнительно). Соответствующие диалоговые окна в Windows XP и Windows Server 2003 немного различаются. Они показаны на рис. 6-17.

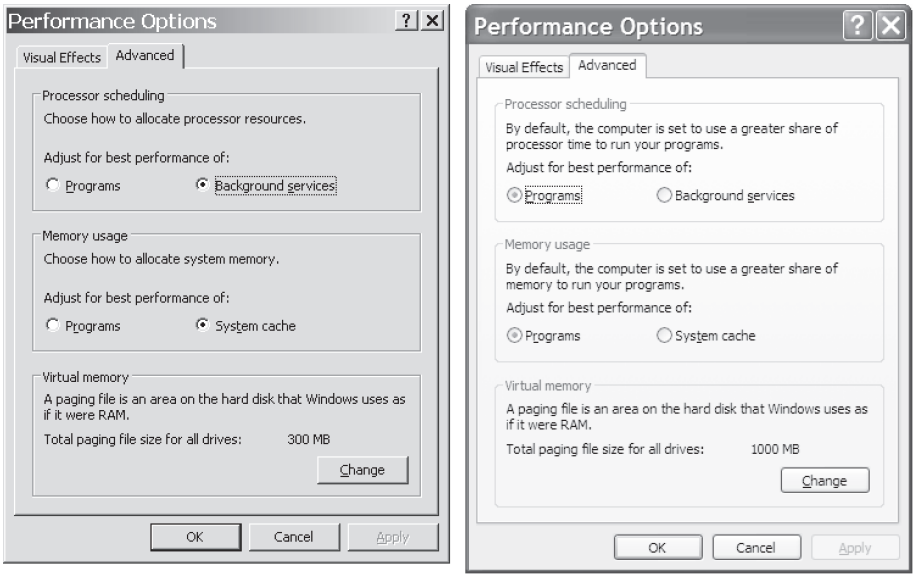


Рис. 6-17. *Задание кванта в Windows XP и Windows Server 2003*

Параметр Programs (Программ), который в Windows 2000 назывался Applications (Приложений), соответствует использованию коротких квантов

переменной величины — этот вариант действует для Windows 2000 Professional и Windows XP по умолчанию. Если вы установите Terminal Services в систему Windows Server и настроите ее как сервер приложений, то и в такой системе будет выбран именно этот параметр, чтобы пользователи сервера терминала получали одинаковые кванты, как и в клиентских или персональных системах. Работая с Windows Server как с персональной операционной системой, вы также могли бы вручную выбрать этот параметр.

Параметр Background Services (Фоновых служб) подразумевает применение длинных квантов фиксированного размера, что предлагается по умолчанию в системах Windows Server. Единственная причина, по которой имело бы смысл выбрать этот параметр на рабочей станции, — ее использование в качестве серверной системы.

Еще одно различие между параметрами Programs и Background Services заключается в том, какой эффект они оказывают на кванты потоков в активном процессе. Об этом рассказывается в следующем разделе.

Динамическое увеличение кванта

До Windows NT 4.0, когда на рабочей станции или в клиентской системе какое-то окно становилось активным, приоритет всех потоков активного процесса (которому принадлежит поток, владеющий окном в фокусе ввода) динамически повышался на 2. Повышенный приоритет действовал до тех пор, пока любому потоку процесса принадлежало активное окно. Проблема с этим подходом была в том, что, если вы запустили длительный процесс, интенсивно использующий процессор (например, начали пересчет электронной таблицы), и переключились на другой процесс, требующий больших вычислительных ресурсов (скажем, на одну из программ CAD, графический редактор или какую-нибудь игру), то первый процесс, ставший теперь фоновым, получит лишь очень малую часть процессорного времени (или вообще не получит его). А все дело в том, что приоритет потоков активного процесса повышается на 2 (здесь предполагается, что базовый приоритет потоков обоих процессов был одинаковым).

Это поведение по умолчанию изменилось с появлением Windows NT 4.0 Workstation — кванты потоков активного процесса стали увеличиваться в 3 раза. Таким образом, по умолчанию на рабочих станциях их квант достигал 6 тактов таймера, а у потоков остальных процессов — 2 тактов. Благодаря этому, когда процесс, интенсивно использующий процессорные ресурсы, оказывается фоновым, новый активный процесс получает пропорционально большее процессорное время (и вновь предполагается, что приоритеты потоков одинаковы как в активном, так и в фоновом процессе).

Заметьте, что это изменение квантов относится лишь к процессам с приоритетом выше Idle в системах с установленным параметром Programs (или Applications в Windows 2000) в диалоговом окне Performance Options (Параметры быстродействия), как пояснялось в предыдущем разделе. Кванты потоков активного процесса в системах с установленным параметром Background Services (настройка по умолчанию в системах Windows Server) не изменяются.

Параметр реестра для настройки кванта

Пользовательский интерфейс, позволяющий изменить относительную величину кванта, модифицирует в реестре параметр HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl\Win32PrioritySeparation. Этот же параметр определяет, можно ли динамически увеличивать (и, если да, то насколько) кванты потоков, выполняемых в активном процессе. Данный параметр содержит 3 двухбитных поля (рис. 6-18).

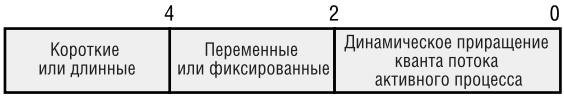


Рис. 6-18. *Поля параметра Win32PrioritySeparation*

- **Короткие или длинные** Значение 1 указывает на длинные кванты, а 2 — на короткие. Если это поле равно 0 или 3, используются кванты по умолчанию (короткие в Windows 2000 Professional или Windows XP и длинные в системах Windows Server).
- **Переменные или фиксированные** Если задано значение 1, кванты потоков активного процесса могут варьироваться, а если задано значение 2 — нет. Если это поле равно 0 или 3, используется настройка по умолчанию (переменные в Windows 2000 Professional или Windows XP и фиксированные в системах Windows Server).
- **Динамическое приращение кванта потока активного процесса** Это поле (хранящееся в переменной ядра *PsPrioritySeparation*) может быть равно 0, 1 или 2 (значение 3 недопустимо и интерпретируется как 2) и представляет собой индекс в трехэлементном байтовом массиве (*PspForegroundQuantum*), используемом для расчета квантов потоков активного процесса. Кванты потоков фоновых процессов определяются первым элементом этого массива. Возможные значения в *PspForegroundQuantum* перечислены в таблице 6-16.

Таблица 6-16. *Величины квантов*

	Короткие			Длинные		
Переменные	6	12	18	12	24	36
Фиксированные	18	18	18	36	36	36

Заметьте, что при использовании диалогового окна Performance Options (Параметры быстродействия) доступны лишь две комбинации: короткие кванты с утроением в активном процессе или длинные без изменения в таком процессе. Но прямое редактирование параметра Win32PrioritySeparation в реестре позволяет выбирать и другие комбинации.

Сценарии планирования

Известно, что вопрос «Какому потоку отдать процессорное время?» Windows 2000 решает, исходя из приоритетов. Но как этот подход работает на прак-

тике? Следующие разделы иллюстрируют, как вытесняющая многозадачность, управляемая на основе приоритетов, действует на уровне потоков.

Самостоятельное переключение

Во-первых, поток может самостоятельно освободить процессор, перейдя в состояние ожидания на каком-либо объекте (например, событии, мьютексе, семафоре, порте завершения ввода-вывода, процессе, потоке, оконном сообщении и др.) путем вызова одной из многочисленных Windows-функций ожидания (скажем, *WaitForSingleObject* или *WaitForMultipleObjects*). Ожидание на объектах было рассмотрено в главе 3.

На рис. 6-19 показано, как поток входит в состояние ожидания и как Windows выбирает новый поток для выполнения.

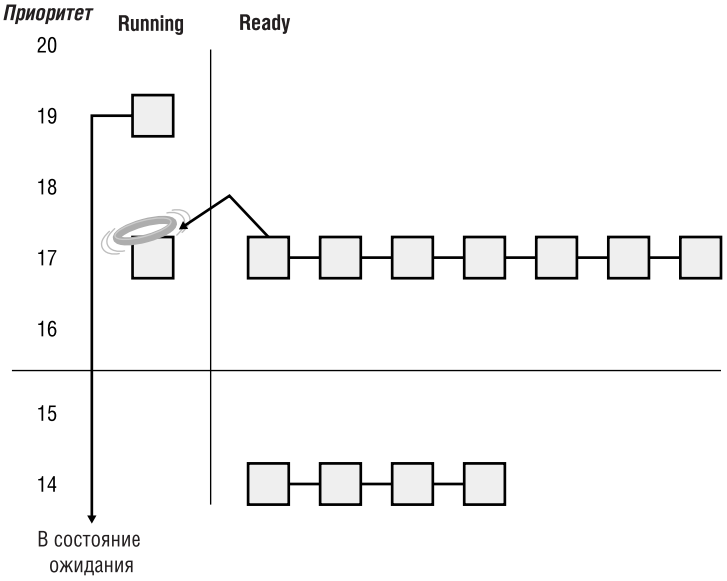


Рис. 6-19. *Самостоятельное переключение*

На рис. 6-19 поток (верхний блок) самостоятельно освобождает процессор, в результате чего к процессору подключается другой поток из очереди (отмеченный кольцом в колонке Running). Исходя из этой схемы, можно подумать, что приоритет потока, освобождающего процессор, снижается, но это не так — он просто переводится в очередь ожидания выбранных им объектов. А что происходит с оставшейся частью кванта этого потока? Когда поток входит в состояние ожидания, квант не сбрасывается. Как уже говорилось, после успешного завершения ожидания квант потока уменьшается на одну единицу, что эквивалентно трети интервала таймера (исключение составляют потоки с приоритетом от 14 и выше, у которых после ожидания квант сбрасывается).

Вытеснение

В этом сценарии поток с более низким приоритетом вытесняется готовым к выполнению потоком с более высоким приоритетом. Такая ситуация может быть следствием двух обстоятельств:

- завершилось ожидание потока с более высоким приоритетом (т. е. произошло событие, которого он ждал);
- приоритет потока увеличился или уменьшился.

В любом из этих случаев Windows решает, продолжить выполнение текущего потока или вытеснить его потоком с более высоким приоритетом.

> **ПРИМЕЧАНИЕ** Потоки пользовательского режима могут вытеснять потоки режима ядра. То есть режим выполнения потока значения не имеет — определяющим фактором является его приоритет.

Когда поток вытесняется, он помещается в начало очереди готовых потоков соответствующего уровня приоритета. Эту ситуацию иллюстрирует рис. 6-20.

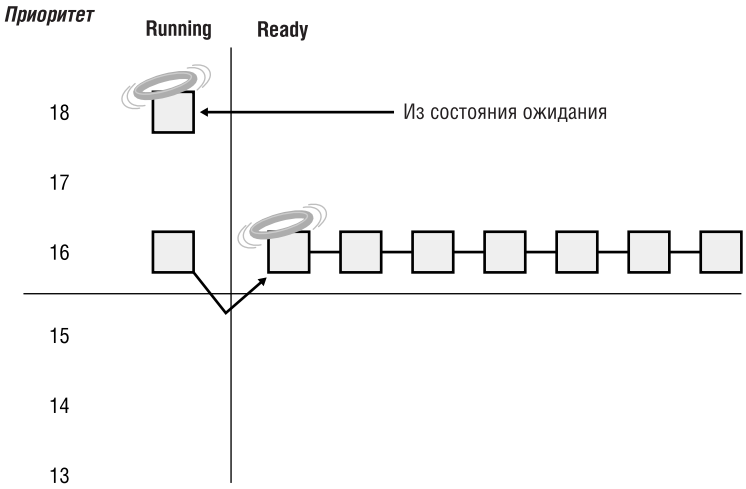


Рис. 6-20. *Планирование потоков с вытеснением*

На рис. 6-20 поток с приоритетом 18 выходит из состояния ожидания и вновь захватывает процессор, вытесняя выполняемый в этот момент поток (с приоритетом 16) в очередь готовых потоков. Заметьте, что вытесненный поток помещается не в конец, а в начало очереди. После завершения вытеснившего потока вытесненный сможет отработать остаток своего кванта.

Завершение кванта

Когда поток израсходует свой квант процессорного времени, Windows должна решить, следует ли понизить его приоритет и подключить к процессору другой поток.

Снизив приоритет потока, Windows ищет более подходящий для выполнения поток (таким потоком, например, будет любой из очереди готовых потоков с приоритетом выше нового приоритета текущего потока). Если Windows оставляет приоритет потока прежним и в очереди готовых потоков есть другие потоки с тем же приоритетом, она выбирает из очереди следующий поток с тем же приоритетом, а выполнявшийся до этого поток перемещает в хвост очереди (задавая ему новую величину кванта и переводя его из состояния Running в состояние Ready). Этот случай иллюстрирует рис. 6-21. Если ни один поток с тем же приоритетом не готов к выполнению, текущему потоку выделяется еще один квант процессорного времени.

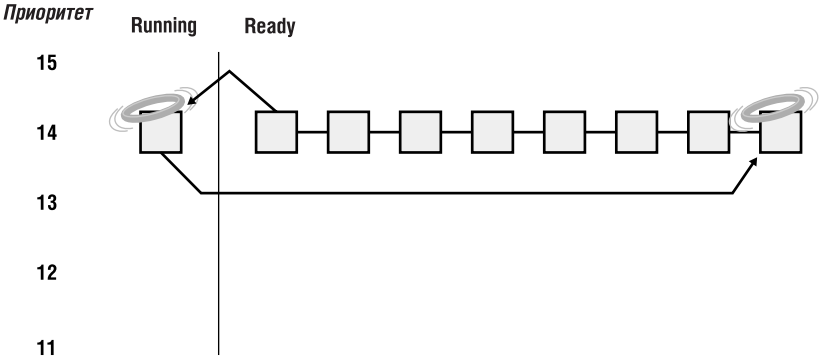


Рис. 6-21. *Планирование потоков в момент завершения кванта текущего потока*

Завершение потока

Завершаясь (после возврата из основной процедуры и вызова *ExitThread* или из-за уничтожения вызовом *TerminateThread*), поток переходит в состояние Terminated. Если в этот момент ни один описатель его объекта «поток» не открыт, поток удаляется из списка потоков процесса, и соответствующие структуры данных освобождаются.

Переключение контекста

Контекст потока и процедура его переключения зависят от архитектуры процессора. В типичном случае переключение контекста требует сохранения и восстановления следующих данных:

- указателя команд;
- указателей на стек ядра и пользовательский стек;
- указателя на адресное пространство, в котором выполняется поток (каталог таблиц страниц процесса).

Ядро сохраняет эту информацию, заталкивая ее в текущий стек ядра, обновляя указатель стека и сохраняя его в блоке KTHREAD потока. Далее указатель стека ядра устанавливается на стек ядра нового потока и загружается контекст этого потока. Если новый поток принадлежит другому процессу, в

специальный регистр процессора загружается адрес его каталога таблиц страниц, в результате чего адресное пространство этого процесса становится доступным (о трансляции адресов см. в главе 7). При наличии отложенной APC ядра запрашивается прерывание IRQL уровня 1. В ином случае управление передается загруженному для нового потока указателю команд, и выполнение этого потока возобновляется.

Поток простоя

Если нет ни одного потока, готового к выполнению на процессоре, Windows подключает к данному процессору поток простоя (процесса Idle). Для каждого процессора создается свой поток простоя.

Разные утилиты для просмотра процессов в Windows по-разному называют процесс Idle. Диспетчер задач и Process Explorer обозначают его как «System Idle Process», Process Viewer — как «Idle», Pstat — как «Idle Process», Process Explode и Tlist — как «System Process», а Qslice — как «SystemProcess». Windows сообщает, что приоритет потока простоя равен 0. Но на самом деле у него вообще нет уровня приоритета, поскольку он выполняется лишь в отсутствие других потоков. (Вспомните, что на нулевом уровне приоритета в Windows работает лишь поток обнуления страниц; см. главу 7.)

Холостой цикл, работающий при IRQL уровня «DPC/dispatch», просто запрашивает задания, например на доставку отложенных DPC или на поиск потоков, подлежащих диспетчеризации. Хотя последовательность работы потока простоя зависит от архитектуры, он все равно выполняет следующие действия.

1. Включает и отключает прерывания (тем самым давая возможность доставить отложенные прерывания).
2. Проверяет, нет ли у процессора незавершенных DPC (см. главу 3). Если таковые есть, сбрасывает отложенное программное прерывание и доставляет эти DPC.
3. Проверяет, выбран ли какой-нибудь поток для выполнения на данном процессоре, и, если да, организует его диспетчеризацию.
4. Вызывает из HAL процедуру обработки процессора в простое (если нужно выполнить какие-либо функции управления электропитанием).

В Windows Server 2003 поток простоя также проверяет наличие потоков, ожидающих выполнения на других процессорах, но об этом пойдет речь в разделе по планированию потоков в многопроцессорных системах.

Динамическое повышение приоритета

Windows может динамически повышать значение текущего приоритета потока в одном из пяти случаев:

- после завершения операций ввода-вывода;
- по окончании ожидания на событии или семафоре исполнительной системы;

- по окончании операции ожидания потоками активного процесса;
- при пробуждении GUI-потоков из-за операций с окнами;
- если поток, готовый к выполнению, задерживается из-за нехватки процессорного времени.

Динамическое повышение приоритета предназначено для оптимизации общей пропускной способности и отзывчивости системы, а также для устранения потенциально «нечестных» сценариев планирования. Однако, как и любой другой алгоритм планирования, динамическое повышение приоритета — не панацея, и от него выигрывают не все приложения.

ПРИМЕЧАНИЕ Windows никогда не увеличивает приоритет потоков в диапазоне реального времени (16–31). Поэтому планирование таких потоков по отношению к другим всегда предсказуемо. Windows считает: тот, кто использует приоритеты реального времени, знает, что делает.

Динамическое повышение приоритета после завершения ввода-вывода

Windows временно повышает приоритет потоков по окончании определенных операций ввода-вывода, поэтому у потоков, ожидавших завершения таких операций, больше шансов немедленно возобновить выполнение и обработать полученные данные. Вспомните: после пробуждения потока оставшийся у него квант уменьшается на одну единицу, так что потоки, ожидавшие завершения ввода-вывода, не получают неоправданных преимуществ. Хотя рекомендованные приращения в результате динамического повышения приоритета определены в заголовочных файлах DDK (ищите строки «#define IO» в Wdm.h или Ntddk.h; эти же значения перечислены в таблице 6-17), реальное приращение определяется драйвером устройства. Именно драйвер устройства указывает — через функцию ядра *IoCompleteRequest* — на необходимость динамического повышения приоритета после выполнения запроса на ввод-вывод. Заметьте, что для запросов на ввод-вывод, адресованных устройствам, которые гарантируют меньшее время отклика, предусматриваются бóльшие приращения приоритета.

Таблица 6-17. *Рекомендованные приращения приоритета*

Устройство	Приращение приоритета
Диск, CD-ROM, параллельный порт, видео	1
Сеть, почтовый ящик, именованный канал, последовательный порт	2
Клавиатура, мышь	6
Звуковая плата	8

Приоритет потока всегда повышается относительно базового, а не текущего уровня. Как показано на рис. 6-22, после динамического повышения приоритета поток в течение одного кванта выполняется с повышенным уровнем приоритета, после чего приоритет снижается на один уровень и потоку выделяется еще один квант. Этот цикл продолжается до тех пор, пока

приоритет не снизится до базового. Поток с более высоким приоритетом все равно может вытеснить поток с повышенным приоритетом, но прерванный поток должен полностью отработать свой квант с повышенным приоритетом до того, как этот приоритет начнет понижаться.

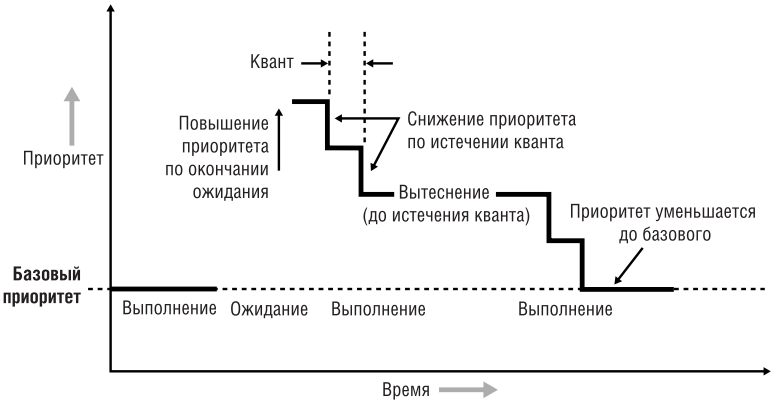


Рис. 6-22. *Динамическое изменение приоритета*

Как уже отмечалось, динамическое повышение приоритета применяется только к потокам с приоритетом динамического диапазона (0–15). Независимо от приращения приоритет потока никогда не будет больше 15. Иначе говоря, если к потоку с приоритетом 14 применить динамическое повышение на 5 уровней, его приоритет возрастет лишь до 15. Если приоритет потока равен 15, он остается неизменным при любой попытке его повышения.

Динамическое повышение приоритета по окончании ожидания событий и семафоров

Когда ожидание потока на событии исполнительной системы или объекте «семафор» успешно завершается (из-за вызова *SetEvent*, *PulseEvent* или *ReleaseSemaphore*), его приоритет повышается на 1 уровень (см. значения EVENT_INCREMENT и SEMAPHORE_INCREMENT в заголовочных файлах DDK). Причина повышения приоритета потоков, закончивших ожидание событий или семафоров, та же, что и для потоков, ожидавших окончания операций ввода-вывода: потокам, блокируемым на событиях, процессорное время требуется реже, чем остальным. Такая регулировка позволяет равномернее распределять процессорное время.

В данном случае действуют те же правила динамического повышения приоритета, что и при завершении операций ввода-вывода (см. предыдущий раздел).

К потокам, которые пробуждаются в результате установки события вызовом специальных функций *NtSetEventBoostPriority* (используется в Ntdll.dll для критических секций) и *KeSetEventBoostPriority* (используется для ресурсов исполнительной системы и блокировок с заталкиванием указателя) повышение приоритета применяется особым образом. Если поток с приоритетом

13 или ниже, ждущий на событии, пробуждается в результате вызова специальной функции, его приоритет повышается до приоритета потока, установившего событие, плюс 1. Если длительность его кванта меньше 4 единиц, она приравнивается 4 единицам. Исходный приоритет восстанавливается по истечении этого кванта.

Динамическое повышение приоритета потоков активного процесса после выхода из состояния ожидания

Всякий раз, когда поток в активном процессе завершает ожидание на объекте ядра, функция ядра *KiUnwaitThread* динамически повышает его текущий (не базовый) приоритет на величину текущего значения *PsPrioritySeparation*. (Какой процесс является активным, определяет подсистема управления окнами.) Как было сказано в разделе «Управление величиной кванта» ранее в этой главе, *PsPrioritySeparation* представляет собой индекс в таблице квантов (байтовом массиве), с помощью которой выбираются величины квантов для потоков активных процессов. Однако в данном случае *PsPrioritySeparation* используется как значение, на которое повышается приоритет.

Это увеличивает отзывчивость интерактивного приложения по окончании ожидания. В результате повышаются шансы на немедленное возобновление его потока — особенно если в фоновом режиме выполняется несколько процессов с тем же базовым приоритетом.

В отличие от других видов динамического повышения приоритета этот поддерживается всеми системами Windows, и его *нельзя* отключить даже через Windows-функцию *SetThreadPriorityBoost*.

ЭКСПЕРИМЕНТ: наблюдение за динамическим изменением приоритета потока активного процесса

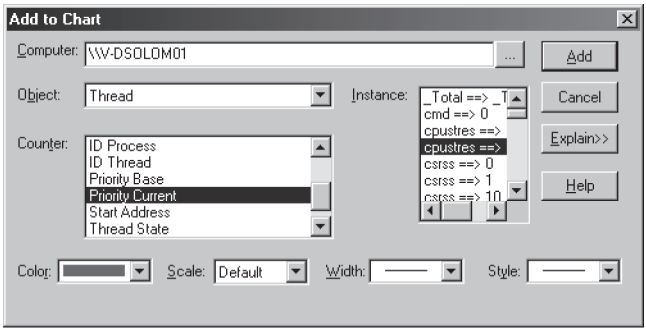
Увидеть механизм повышения приоритета в действии позволяет утилита CPU Stress (входит в состав ресурсов и Platform SDK).

1. В окне Control Panel (Панель управления) откройте апплет System (Система) или щелкните правой кнопкой мыши значок My Computer (Мой компьютер), выберите команду Properties (Свойства) и перейдите на вкладку Advanced (Дополнительно). Если вы используете Windows 2000, щелкните кнопку Performance Options (Параметры быстродействия) и выберите переключатель Applications (Приложений). В случае Windows XP или Windows Server 2003 щелкните кнопку Options (Параметры) в разделе Performance (Быстродействие), откройте вкладку Advanced (Дополнительно) и выберите переключатель Programs (Программ). В итоге *PsPrioritySeparation* получит значение 2.
2. Запустите Cpustres.exe.
3. Запустите Windows NT 4 Performance Monitor (Perfmon4.exe на компакт-диске ресурсов Windows 2000). Для эксперимента нужна имен-

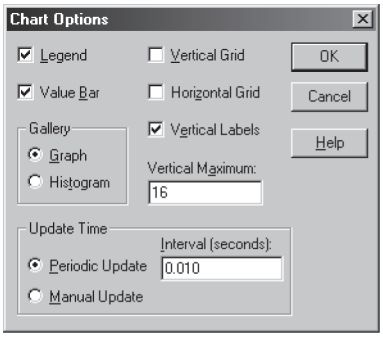
см. след. стр.

но эта устаревшая версия, поскольку она способна запрашивать значения счетчиков производительности с более высокой частотой, чем оснастка Performance (Производительность), которая запрашивает такие значения не чаще, чем раз в секунду.

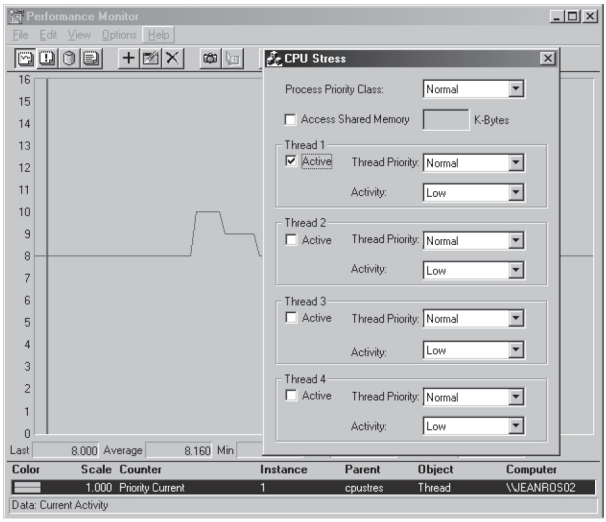
4. Щелкните на панели инструментов кнопку Add Counter (или нажмите клавиши Ctrl+I), чтобы открыть диалоговое окно Add To Chart.
5. Выберите объект Thread и счетчик Priority Current.
6. Пролистайте список Instance и найдите процесс Cpustres. Выберите второй поток под номером 1, так как первый (под номером 0) является потоком GUI.



7. Щелкните кнопку Add, затем — кнопку Done.
8. Из меню Options выберите команду Chart. Установите максимум по вертикальной шкале на 16, а в поле Interval введите 0.010 и щелкните кнопку OK.



9. Теперь активизируйте процесс Cpustres. В результате приоритет потока Cpustres должен повыситься на 2 уровня, а потом снизиться до базового, как показано на следующей иллюстрации.



10. Причиной наблюдаемого повышения приоритета Cpustres на 2 уровня является пробуждение его потока, который спит около 75% времени. Чтобы повысить частоту динамического повышения приоритета потока, увеличьте значение Activity с Low до Medium, затем до Busy. Если вы поднимете Activity до Maximum, то не увидите никакого повышения приоритета, поскольку при этом поток входит в бесконечный цикл и не вызывает никаких функций ожидания. А значит, его приоритет будет оставаться неизменным.
11. Закончив эксперимент, закройте Performance Monitor и CPU Stress.

Динамическое повышение приоритета после пробуждения GUI-потоков

Приоритет потоков, владеющих окнами, дополнительно повышается на 2 уровня после их пробуждения из-за активности подсистемы управления окнами, например при получении оконных сообщений. Подсистема управления окнами (Win32k.sys) повышает приоритет, вызывая *KeSetEvent* для установки события, пробуждающего GUI-поток. Приоритет повышается по той же причине, что и в предыдущем случае, — для создания преимуществ интерактивным приложениям.

ЭКСПЕРИМЕНТ: наблюдаем динамическое повышение приоритета GUI-потоков

Чтобы увидеть, как подсистема управления окнами повышает на 2 уровня приоритет GUI-потоков, пробуждаемых для обработки оконных сообщений, понаблюдайте за текущим приоритетом GUI-приложения, перемещая мышь в пределах его окна. Для этого сделайте вот что.

1. В окне Control Panel (Панель управления) откройте апплет System (Система) или щелкните правой кнопкой мыши значок My Compu-

см. след. стр.

ter (Мой компьютер), выберите команду Properties (Свойства) и перейдите на вкладку Advanced (Дополнительно). Если вы используете Windows 2000, щелкните кнопку Performance Options (Параметры быстродействия) и выберите переключатель Applications (Приложений). В случае Windows XP или Windows Server 2003 щелкните кнопку Options (Параметры) в разделе Performance (Быстродействие), откройте вкладку Advanced (Дополнительно) и выберите переключатель Programs (Программы). В итоге *PsPrioritySeparation* получит значение 2.

2. Запустите Notepad, выбрав из меню Start (Пуск) команды Programs (Программы), Accessories (Стандартные) и Notepad (Блокнот).
3. Запустите Windows NT 4 Performance Monitor (Perfmon4.exe на компакт-диске ресурсов Windows 2000). Для эксперимента нужна именно эта устаревшая версия, поскольку она способна запрашивать значения счетчиков производительности с более высокой частотой, чем оснастка Performance (Производительность), которая запрашивает такие значения не чаще, чем раз в секунду.
4. Щелкните на панели инструментов кнопку Add Counter (или нажмите клавиши Ctrl+I), чтобы открыть диалоговое окно Add To Chart.
5. Выберите объект Thread и счетчик Priority Current.
6. Пролистайте список Instance и найдите процесс Notepad. Выберите поток 0, щелкните кнопку Add, а затем — кнопку Done.
7. Как и в предыдущем эксперименте, выберите из меню Options команду Chart. Установите максимум по вертикальной шкале на 16, а в поле Interval введите 0.010 и щелкните кнопку OK.
8. В итоге вы должны увидеть, как колеблется приоритет нулевого потока Notepad (от 8 до 10). Поскольку Notepad — вскоре после повышения его приоритета (как потока активного процесса) на 2 уровня — перешел в состояние ожидания, его приоритет мог не успеть снизиться с 10 до 9 или до 8.
9. Активизировав окно Performance Monitor, подвигайте курсор мыши в окне Notepad (но сначала расположите эти окна на рабочем столе так, чтобы они оба были видны). Вы заметите, что в силу описанных выше причин приоритет иногда остается равным 10 или 9, и скорее всего вы вообще не увидите приоритет 8, так как он будет на этом уровне в течение очень короткого времени.
10. Теперь сделайте активным окно Notepad. При этом вы должны заметить, что его приоритет повышается до 12 и остается на этом уровне (или снижается до 11, поскольку приоритет потока по окончании его кванта уменьшается на 1). Почему приоритет потока Notepad достигает такого значения? Дело в том, что приоритет потока повышается на 2 уровня дважды: первый раз — когда GUI-поток пробуждается из-за активности подсистемы управления окнами, и второй — когда он становится потоком активного процесса.

11. Если после этого вы снова подвигаете курсор мыши в окне Notepad (пока оно активно), то, возможно, заметите падение приоритета до 11 (или даже до 10) из-за динамического снижения приоритета потока по истечении кванта. Но приоритет этого потока все равно превышает базовый на 2 уровня, так как процесс Notepad остается активным до тех пор, пока активно его окно.
12. Закончив эксперимент, закройте Performance Monitor и Notepad.

Динамическое повышение приоритета при нехватке процессорного времени

Представьте себе такую ситуацию: поток с приоритетом 7 постоянно вытесняет поток с приоритетом 4, не давая ему возможности получить процессорное время; при этом поток с приоритетом 11 ожидает какой-то ресурс, заблокированный потоком с приоритетом 4. Но, поскольку поток с приоритетом 7 занимает все процессорное время, поток с приоритетом 4 никогда не получит процессорное время, достаточное для завершения и освобождения ресурсов, нужных потоку с приоритетом 11. Что же делает Windows в подобной ситуации? Раз в секунду диспетчер настройки баланса (balance set manager), системный поток, предназначенный главным образом для выполнения функций управления памятью (см. главу 7), сканирует очереди готовых потоков и ищет потоки, которые находятся в состоянии Ready в течение примерно 4 секунд. Обнаружив такой поток, диспетчер настройки баланса повышает его приоритет до 15. В Windows 2000 и Windows XP квант потока удваивается относительно кванта процесса. В Windows Server 2003 квант устанавливается равным 4 единицам. Как только квант истекает, приоритет потока немедленно снижается до исходного уровня. Если этот поток не успел закончить свою работу и если другой поток с более высоким приоритетом готов к выполнению, то после снижения приоритета он возвращается в очередь готовых потоков. В итоге через 4 секунды его приоритет может быть снова повышен.

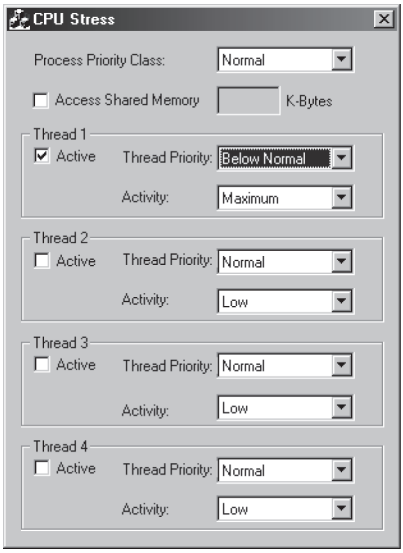
На самом деле диспетчер настройки баланса не сканирует при каждом запуске все потоки, готовые к выполнению. Чтобы свести к минимуму расход процессорного времени, он сканирует лишь 16 готовых потоков. Если таких потоков с данным уровнем приоритета более 16, он запоминает тот поток, перед которым он остановился, и в следующий раз продолжает сканирование именно с него. Кроме того, он повышает приоритет не более чем у 10 потоков за один проход. Обнаружив 10 потоков, приоритет которых следует повысить (что говорит о необычайно высокой загруженности системы), он прекращает сканирование. При следующем проходе сканирование возобновляется с того места, где оно было прервано в прошлый раз.

Всегда ли данный алгоритм решает проблемы, связанные с приоритетами? Вовсе нет — он тоже не совершенен. Но со временем потоки, страдающие от нехватки процессорного времени, обязательно получают время, достаточное для завершения обработки текущих данных и возврата в состояние ожидания.

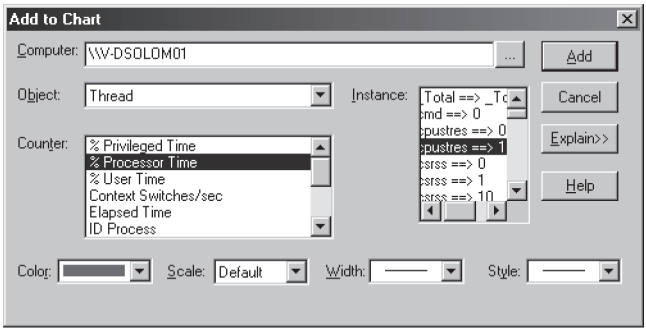
ЭКСПЕРИМЕНТ: динамическое повышение приоритетов при нехватке процессорного времени

Утилита CPU Stress (она входит в состав ресурсов и Platform SDK) позволяет наблюдать, как работает механизм динамического повышения приоритетов. В этом эксперименте мы увидим, как изменяется интенсивность использования процессора при повышении приоритета потока. Для этого проделайте следующие операции.

1. Запустите Cpustres.exe. Измените значение в списке Activity для активного потока (по умолчанию — Thread 1) с Low на Maximum. Далее смените приоритет потока с Normal на Below Normal. При этом окно утилиты должно выглядеть так:

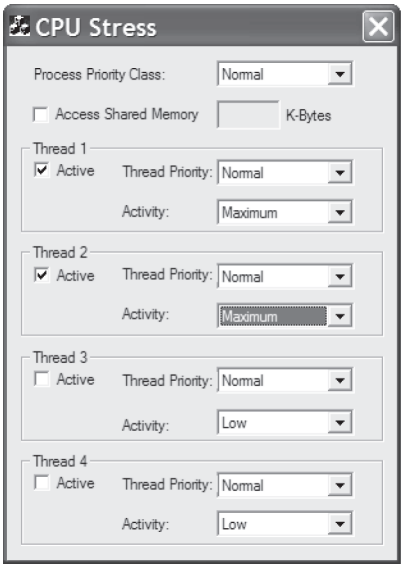


2. Запустите Windows NT 4 Performance Monitor (Perfmon4.exe на компакт-диске ресурсов Windows 2000). Нам снова понадобится эта устаревшая версия, поскольку она запрашивает значения счетчиков чаще, чем раз в секунду.
3. Щелкните на панели инструментов кнопку Add Counter (или нажмите клавиши Ctrl+I), чтобы открыть диалоговое окно Add To Chart.
4. Выберите объект Thread и счетчик % Processor Time.
5. Пролистайте список Instance и найдите процесс Cpustres. Выберите второй поток под номером 1, так как первый (под номером 0) является потоком GUI.



6. Щелкните кнопку Add, затем — кнопку Done.
7. Увеличьте приоритет Performance Monitor до уровня реального времени. Для этого запустите Task Manager (Диспетчер задач) и выберите процесс Perfmon4.exe на вкладке Processes (Процессы). Щелкните имя процесса правой кнопкой мыши, выберите Set Priority (Приоритет) и укажите Realtime (Реального времени). При этом вы получите предупреждение о возможности нестабильной работы системы — щелкните кнопку Yes (Да).
8. Запустите еще один экземпляр CPU Stress. Измените в нем параметр Activity для Thread 1 с Low на Maximum.
9. Теперь переключитесь обратно в Performance Monitor. Вы должны наблюдать всплески активности процессора примерно каждые 4 секунды, так как приоритет потока возрос до 15.

Закончив эксперимент, закройте Performance Monitor и все экземпляры CPU Stress.



ЭКСПЕРИМЕНТ: «прослушивание» динамического повышения приоритета

Чтобы «услышать» эффект динамического повышения приоритета потока при нехватке процессорного времени, в системе со звуковой платой выполните следующие операции.

1. Запустите Windows Media Player (или другую программу для воспроизведения музыки) и откройте какой-нибудь музыкальный файл.
2. Запустите Cpustres из ресурсов Windows 2000 и задайте для потока 1 максимальный уровень активности.
3. Повысьте приоритет потока 1 с Normal до Time Critical.
4. Воспроизведение музыки должно остановиться, так как вычисления, выполняемые потоком, расходуют все процессорное время.
5. Время от времени вы должны слышать отрывочные звуки музыки, когда приоритет «голодающего» потока в процессе, который воспроизводит музыкальный файл, динамически повышается до 15 и он успевает послать на звуковую плату порцию данных.
6. Закройте Cpustres и Windows Media Player.

Многопроцессорные системы

В однопроцессорной системе алгоритм планирования относительно прост: всегда выполняется поток с наивысшим приоритетом, готовый к выполнению. В многопроцессорной системе планирование усложняется, так как Windows пытается подключать поток к наиболее оптимальному для него процессору, учитывая предпочтительный и предыдущий процессоры для этого потока, а также конфигурацию многопроцессорной системы. Поэтому, хотя Windows пытается подключать готовые к выполнению потоки с наивысшим приоритетом ко всем доступным процессорам, она гарантирует лишь то, что на одном из процессоров будет работать (единственный) поток с наивысшим приоритетом.

Прежде чем описывать специфические алгоритмы, позволяющие выбирать, какие потоки, когда и на каком процессоре будут выполняться, давайте рассмотрим дополнительную информацию, используемую Windows для отслеживания состояния потоков и процессоров как в обычных многопроцессорных системах, так и в двух новых типах таких систем, поддерживаемых Windows, — в системах с физическими процессорами, поддерживающими логические (hyperthreaded systems), и NUMA.

База данных диспетчера ядра в многопроцессорной системе

Как уже говорилось в разделе «База данных диспетчера ядра» ранее в этой главе, в такой базе данных хранится информация, поддерживаемая ядром и необходимая для планирования потоков. В многопроцессорных системах Windows 2000 и Windows XP (рис. 6-15) очереди готовых потоков и сводка готовых потоков имеют ту же структуру, что и в однопроцессорных систе-

мах. Кроме того, Windows поддерживает две битовые маски для отслеживания состояния процессоров в системе. (Как используются эти маски, см. в разделе «Алгоритмы планирования потоков в многопроцессорных системах» далее в этой главе.) Вот что представляют собой эти маски.

- *Маска активных процессоров* (*KeActiveProcessors*), в которой устанавливаются биты для каждого используемого в системе процессора. (Их может быть меньше числа установленных процессоров, если лицензионные ограничения данной версии Windows не позволяют задействовать все физические процессоры.)
- *Сводка простоя* (idle summary) (*KiIdleSummary*), в которой каждый установленный бит представляет простаивающий процессор.

Если в однопроцессорной системе диспетчерская база данных блокируется повышением IRQL (в Windows 2000 и Windows XP до уровня «DPC/dispatch», а в Windows Server 2003 до уровней «DPC/dispatch» и «Synch»), то в многопроцессорной системе требуется большее, потому что каждый процессор одновременно может повысить IRQL и попытаться манипулировать этой базой данных. (Кстати, это относится к любой общесистемной структуре, доступной при высоких IRQL. Общее описание синхронизации режима ядра и спин-блокировок см. в главе 3.) В Windows 2000 и Windows XP для синхронизации доступа к информации о диспетчеризации потока применяется две спин-блокировки режима ядра: *спин-блокировка диспетчера ядра* (dispatcher spinlock) (*KiDispatcherLock*) и *спин-блокировка обмена контекста* (context swap spinlock) (*KiContextSwapLock*). Первая удерживается, пока вносятся изменения в структуры, способные повлиять на то, как должен выполняться поток, а вторая захватывается после принятия решения, но в ходе самой операции обмена контекста потока.

Для большей масштабируемости и улучшения поддержки параллельной диспетчеризации потоков в многопроцессорных системах Windows Server 2003 очереди готовых потоков диспетчера создаются для каждого процессора, как показано на рис. 6-23. Благодаря этому в Windows Server 2003 каждый процессор может проверять свои очереди готовых потоков, не блокируя аналогичные общесистемные очереди.

Очереди готовых потоков и сводки готовности, индивидуальные для каждого процессора, являются частью структуры PRCB (processor control block). (Чтобы увидеть поля этой структуры, введите **dt nt!_prcb** в отладчике ядра.) Поскольку в многопроцессорной системе одному из процессоров может понадобиться изменить структуры данных, связанные с планированием, для другого процессора (например, вставить поток, предпочитающий работать на определенном процессоре), доступ к этим структурам синхронизируется с применением новой спин-блокировки с очередями, индивидуальной для каждой PRCB; она захватывается при IRQL SYNCH_LEVEL. (Возможные значения SYNCH_LEVEL см. в таблице 6-18.) Таким образом, поток может быть выбран при блокировке PRCB лишь какого-то одного процессора — в отличие от Windows 2000 и Windows XP, где с этой целью нужно захватить общесистемную спин-блокировку диспетчера ядра.

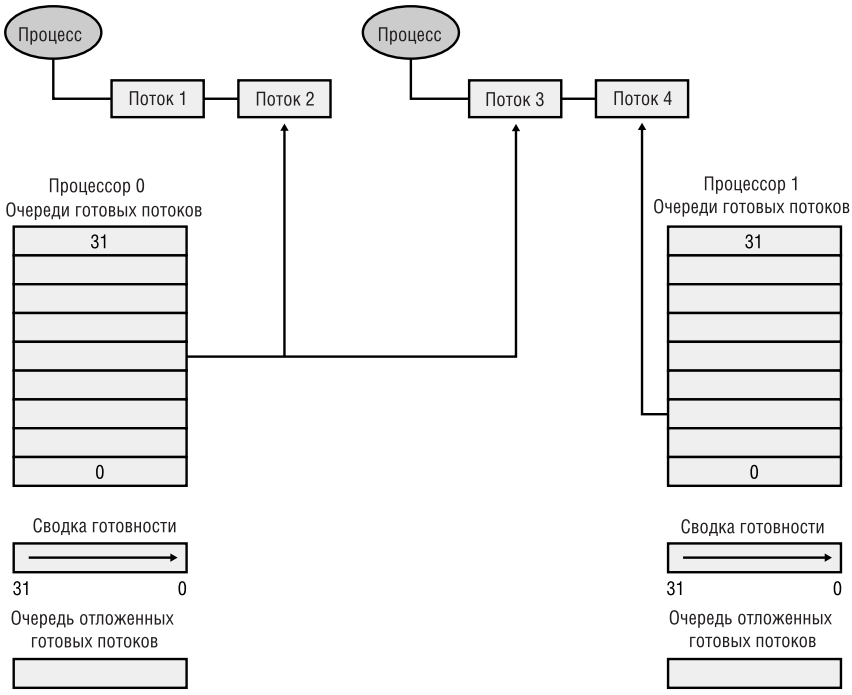


Рис. 6-23. *База данных диспетчера ядра в многопроцессорной системе Windows Server 2003*

Таблица 6-18. *IRQL SYNCH_LEVEL в многопроцессорных системах*

Версия Windows	Количество процессоров
Windows 2000	2
Windows XP на x86	2
Windows Server 2003 на x86	27
Windows XP на x64	12
Windows XP на IA-64	2
Windows Server 2003 на x64 и IA-64	12

Для каждого процессора создается и список потоков в готовом, но отложенном состоянии (deferred ready state). Это потоки, готовые к выполнению, но операция, уведомляющая в результате об их готовности, отложена до более подходящего времени. Поскольку каждый процессор манипулирует только своим списком отложенных готовых потоков, доступ к этому списку не синхронизируется по спин-блокировке PRCB. Список отложенных готовых потоков обрабатывается до выхода из диспетчера потоков, до переключения контекста и после обработки DPC. Потоки в этом списке либо немедленно диспетчеризуются, либо перемещаются в одну из индивидуальных для каждого процессора очередей готовых потоков (в зависимости от приоритета).

Заметьте, что общесистемная спин-блокировка диспетчера ядра по-прежнему существует и используется в Windows Server 2003, но она захватывается лишь на период, необходимый для модификации общесистемного состояния, которое может повлиять на то, какой поток будет выполняться следующим. Например, изменения в синхронизирующих объектах (мьютексах, событиях и семафорах) и их очередях ожидания требуют захвата блокировки диспетчера ядра, чтобы предотвратить попытки модификации таких объектов (и последующей операции перевода потоков в состояние готовности к выполнению) более чем одним процессором. Другие примеры — изменение приоритета потока, срабатывание таймера и обмен содержимого стеков ядра для потоков.

Наконец, в Windows Server 2003 улучшена сихронизация переключения контекста потоков, так как теперь оно синхронизируется с применением спин-блокировки, индивидуальной для каждого потока, а в Windows 2000 и Windows XP переключение контекста синхронизировалось захватом общесистемной спин-блокировки обмена контекста.

Системы с поддержкой Hyperthreading

Как уже говорилось в разделе «Симметричная многопроцессорная обработка» главы 2, Windows XP и Windows Server 2003 поддерживают многопроцессорные системы, использующие технологию Hyperthreading (аппаратная реализация логических процессоров на одном физическом).

1. Логические процессоры не подпадают под лицензионные ограничения на число физических процессоров. Так, Windows XP Home Edition, которая по условиям лицензии может использовать только один процессор, задействует оба логических процессора в однопроцессорной системе с поддержкой Hyperthreading.
2. Если все логические процессоры какого-либо физического процессора простаивают, для выполнения потока выбирается один из логических процессоров этого физического процессора, а не того, у которого один из логических процессоров уже выполняет другой поток.

ЭКСПЕРИМЕНТ: просмотр информации, связанной с Hyperthreading

Изучить такую информацию позволяет команда *!smt* отладчика ядра. Следующий вывод получен в системе с двумя процессорами Xeon с технологией Hyperthreading (четыре логических процессора):

```
lkd> !smt
SMT Summary:
------
   KeActiveProcessors: ****-------------- (0000000f)
        KiIdleSummary: -***-------------- (0000000e)
No PRCB     Set Master SMT Set                                    #LP IAID
 0 ffdff120 Master     *-*-------------- (00000005)   2  00
 1 f771f120 Master     -*-*-------------- (0000000a)   2  06
```

см. след. стр.

```
  2 f7727120 ffdff120   *-*-------------- (00000005)   2  01
  3 f772f120 f771f120   -*-*-------------- (0000000a)   2  07

    Number of licensed physical processors: 2
```

Логические процессоры 0 и 1 находятся на разных физических процессорах (на что указывает ключевое слово «Master»).

Системы NUMA

Другой тип многопроцессорных систем, поддерживаемый Windows XP и Windows Server 2003, — архитектуры памяти с неунифицированным доступом (nonuniform memory access, NUMA). В NUMA-системе процессоры группируются в узлы. В каждом узле имеются свои процессоры и память, и он подключается к системе соединительной шиной с когерентным кэшем (cache-coherent interconnect bus). Доступ к памяти в таких системах называется неунифицированным потому, что у каждого узла есть локальная высокоскоростная память. Хотя любой процессор в любом узле может обращаться ко всей памяти, доступ к локальной для узла памяти происходит гораздо быстрее.

Ядро поддерживает информацию о каждом узле в NUMA-системе в структурах данных KNODE. Переменная ядра *KeNodeBlock* содержит массив указателей на структуры KNODE для каждого узла. Формат структуры KNODE можно просмотреть командой *dt* отладчика ядра:

```
lkd> dt nt!_knode
nt!_KNODE
   +0x000 ProcessorMask    : Uint4B
   +0x004 Color            : Uint4B
   +0x008 MmShiftedColor   : Uint4B
   +0x00c FreeCount        : [2] Uint4B
   +0x018 DeadStackList    : _SLIST_HEADER
   +0x020 PfnDereferenceSListHead : _SLIST_HEADER
   +0x028 PfnDeferredList  : Ptr32 _SINGLE_LIST_ENTRY
   +0x02c Seed             : UChar
   +0x02d Flags            : _flags
```

ЭКСПЕРИМЕНТ: просмотр информации, относящейся к NUMA

Вы можете исследовать информацию, поддерживаемую Windows для каждого узла в NUMA-системе, с помощью команды *!numa* отладчика ядра. Ниже приведен фрагмент вывода, полученный в 32-процессорной NUMA-системе производства NEC с 4 процессорами в каждом узле:

```
21: kd> !numa
NUMA Summary:
------
    Number of NUMA nodes : 8
    Number of Processors : 32
```

```
    MmAvailablePages     : 0x00F70D2C
    KeActiveProcessors   : ********************************----------------
(00000000ffffffff)

    NODE 0 (E00000008428AE00):
        ProcessorMask    : ****------------------------------
        Color            : 0x00000000
        MmShiftedColor   : 0x00000000
        Seed             : 0x00000000
        Zeroed Page Count: 0x00000000001CF330
        Free Page Count  : 0x0000000000000000

    NODE 1 (E00001597A9A2200):
        ProcessorMask    : --****----------------------------
        Color            : 0x00000001
        MmShiftedColor   : 0x00000040
        Seed             : 0x00000006
        Zeroed Page Count: 0x00000000001F77A0
        Free Page Count  : 0x0000000000000004
```

А это фрагмент вывода, полученный в 64-процессорной NUMA-системе производства Hewlett Packard с 4 процессорами в каждом узле:

```
26: kd> !numa
NUMA Summary:
------
    Number of NUMA nodes : 16
    Number of Processors : 64
    MmAvailablePages     : 0x03F55E67
    KeActiveProcessors   :
************************************************************
** (ffffffffffffffff)

    NODE 0 (E000000084261900):
        ProcessorMask    : ****------------------------------
        Color            : 0x00000000
        MmShiftedColor   : 0x00000000
        Seed             : 0x00000001
        Zeroed Page Count: 0x00000000003F4430
        Free Page Count  : 0x0000000000000000

    NODE 1 (E0000145FF992200):
        ProcessorMask    : --****----------------------------
        Color            : 0x00000001
        MmShiftedColor   : 0x00000040
        Seed             : 0x00000007
        Zeroed Page Count: 0x00000000003ED59A
        Free Page Count  : 0x0000000000000000
```

Приложения, которым нужно выжать максимум производительности из NUMA-систем, могут устанавливать маски привязки процесса к процессорам в определенном узле. Получить эту информацию позволяют функции, перечисленные в таблице 6-19. (Функции, с помощью которых можно изменять привязку потоков к процессорам, были перечислены в таблице 6-14.)

Таблица 6-19. *Функции, связанные с NUMA*

Функция	Описание
GetNumaHighestNodeNumber	Возвращает узел с наивысшим в данный момент номером
GetNumaNodeProcessorMask	Возвращает маску процессоров для указанного узла
GetNumaProcessorNode	Возвращает номер узла для указанного процессора

О том, как алгоритмы планирования учитывают особенности NUMA-систем, см. в разделе «Алгоритмы планирования потоков в многопроцессорных системах» далее в этой главе (а об оптимизациях в диспетчере памяти для использования преимуществ локальной для узла памяти см. в главе 7).

Привязка к процессорам

У каждого потока есть *маска привязки к процессорам* (affinity mask), указывающая, на каких процессорах можно выполнять данный поток. Потоки наследуют маску привязки процесса. По умолчанию начальная маска для всех процессов (а значит, и для всех потоков) включает весь набор активных процессоров в системе, т. е. любой поток может выполняться на любом процессоре.

Однако для повышения пропускной способности и/или оптимизации рабочих нагрузок на определенный набор процессоров приложения могут изменять маску привязки потока к процессорам. Это можно сделать на нескольких уровнях.

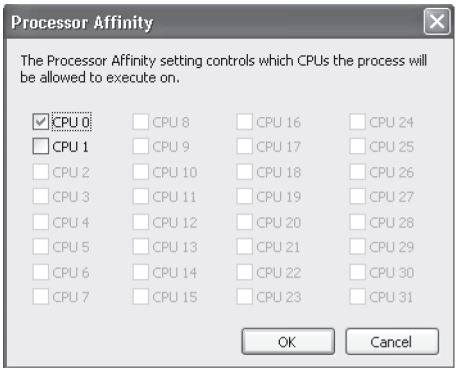
- Вызовом функции *SetThreadAffinityMask*, чтобы задать маску привязки к процессорам для индивидуального потока;
- Вызовом функции *SetProcessAffinityMask*, чтобы задать маску привязки к процессорам для всех потоков в процессе. Диспетчер задач и Process Explorer предоставляют GUI-интерфейс к этой функции: щелкните процесс правой кнопкой мыши и выберите Set Affinity (Задать соответствие). Утилита Psexec (с сайта *www.sysinternals.com*) предоставляет к той же функции интерфейс командной строки (см. ключ *–a*).
- Включением процесса в задание, в котором действует глобальная для задания маска привязки к процессорам, установленная через функцию *SetInformationJobObject* (о заданиях см. раздел «Объекты-задания» далее в этой главе.)
- Определением маски привязки к процессорам в заголовке образа с помощью, например, утилиты Imagecfg из Windows 2000 Server Resource Kit Supplement 1. (О формате образов в Windows см. статью «Portable Executable and Common Object File Format Specification» в MSDN Library.)

В образе можно установить и «однопроцессорный» флаг (используя в Imagecfg ключ *–u*). Если этот флаг установлен, система выбирает один процессор в момент создания процесса и закрепляет его за этим процессом; при этом процессоры меняются от первого и до последнего по принципу карусели. Например, в двухпроцессорной системе при первом запуске образа, помеченного как однопроцессорный, он закрепляется за процессором 0, при втором — за процессором 1, при третьем — за процессором 0, при четвертом — за процессором 1 и т. д. Этот флаг полезен, когда нужно временно обойти ошибку в программе, связанную с неправильной синхронизацией потоков, но проявляющуюся только в многопроцессорных системах.

ЭКСПЕРИМЕНТ: просмотр и изменение привязки процесса к процессорам

В этом эксперименте вы модифицируете привязку процесса к процессорам и убедитесь, что привязка наследуется новыми процессами.

1. Запустите окно командной строки (cmd.exe).
2. Запустите диспетчер задач или Process Explorer и найдите cmd.exe в списке процессов.
3. Щелкните этот процесс правой кнопкой мыши и выберите команду Set Affinity (Задать соответствие). Должен появиться список процессоров. Например, в двухпроцессорной системе вы увидите окно, как на следующей иллюстрации.



4. Выберите подмножество доступных процессоров в системе и нажмите OK. Теперь потоки процесса будут работать только на выбранных вами процессорах.
5. Запустите Notepad.exe из окна командной строки (набрав **notepad.exe**).
6. Вернитесь в диспетчер задач или Process Explorer и найдите новый процесс Notepad. Щелкните его правой кнопкой мыши и выберите Set Affinity. Вы должны увидеть список процессоров, выбранных вами для процесса cmd.exe. Это вызвано тем, что процессы наследуют привязки к процессорам от своего родителя.

ЭКСПЕРИМЕНТ: изменение маски привязки образа

В этом эксперименте (который потребует доступа к многопроцессорной системе) вы измените маску привязки к процессорам для какой-нибудь программы, чтобы заставить ее работать на первом процессоре.

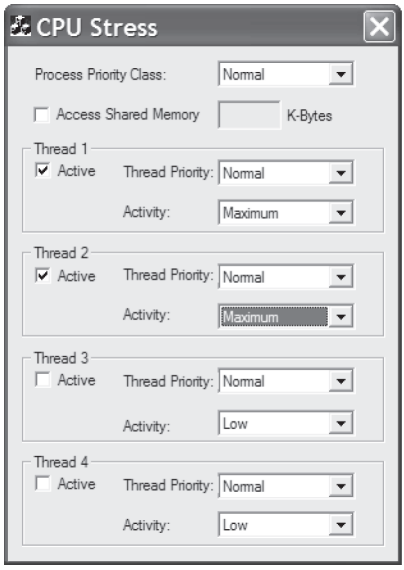
1. Создайте копию Cpustres.exe (эта утилита содержится в ресурсах Windows 2000). Например, если у вас есть каталог c:\temp, то в командной строке введите:

   ```
   copy c:\program files\resource kit\cpustres.exe c:\temp\cpustres.exe
   ```

2. Задайте маску привязки так, чтобы она заставляла потоки процесса выполняться на процессоре 0. Для этого в командной строке (предполагается, что путь к ресурсам прописан в переменной окружения PATH) введите:

   ```
   imagecfg -a 1 c:\temp\cpustres.exe
   ```

3. Теперь запустите модифицированную Cpustres из каталога c:\temp.
4. Включите два рабочих потока и установите уровень активности обоих потоков в Maximum (не Busy). Окно Cpustres должно выглядеть следующим образом.



5. Найдите процесс Cpustres в Process Explorer или диспетчере задач, щелкните его правой кнопкой мыши и выберите Set Affinity. Вы должны увидеть, что процесс привязан к процессору 0.
6. Посмотрите общесистемное использование процессора, выбрав Show, System Information (в Process Explorer) или открыв вкладку Performance (в диспетчере задач). Если в системе нет других процессов, занятых интенсивными вычислениями, общая процентная

доля использования процессорного времени должна составить примерно 1/(число процессоров) (скажем, около 50% в двухпроцессорной системе или около 25% в четырехпроцессорной), потому что оба потока в Cpustres привязаны к одному процессору и остальные процессоры простаивают.

7. Наконец, измените маску привязки процесса Cpustres так, чтобы разрешить ему выполнение на всех процессорах. Снова проверьте общесистемное использование процессорного времени. Теперь оно должно быть 100% в двухпроцессорной системе, 50% в четырехпроцессорной и т. д.

Windows не перемещает уже выполняемый поток с одного процессора на второй, чтобы готовый поток с маской привязки именно к первому процессору немедленно начал на нем работать. Рассмотрим, например, такой сценарий: к процессору 0 подключен поток с приоритетом 8, который может работать на любом процессоре, а к процессору 1 — поток с приоритетом 4, который тоже может выполняться на любом процессоре. Но вот готов поток с приоритетом 6, привязанный только к процессору 0. Что произойдет? Windows не станет переключать поток с приоритетом 8 на процессор 1 (вытесняя при этом поток с приоритетом 4), и поток с приоритетом 6 будет ждать освобождения процессора 0.

Следовательно, изменение маски привязки для процесса или потока может привести к тому, что потоки будут получать меньше процессорного времени, чем обычно, поскольку это ограничивает Windows в выборе процессоров для данного потока. А значит, задавать маску привязки нужно с крайней осторожностью — в большинстве ситуаций оптимальнее оставить выбор за самой Windows.

Идеальный и последний процессоры

В блоке потока ядра каждого потока хранятся номера двух особых процессоров:

- *идеального* (ideal processor) — предпочтительного для выполнения данного потока;
- *последнего* (last processor) — на котором поток работал в прошлый раз.

Идеальный процессор для потока выбирается случайным образом при его создании с использованием зародышевого значения (seed) в блоке процесса. Это значение увеличивается на 1 всякий раз, когда создается новый поток, поэтому создаваемые потоки равномерно распределяются по набору доступных процессоров. Например, первый поток в первом процессе в системе закрепляется за идеальным процессором 0, второй поток того же процесса — за идеальным процессором 1. Однако у следующего процесса в системе идеальный процессор для первого потока устанавливается в 1, для второго — в 2 и т. д. Благодаря этому потоки внутри каждого процесса равномерно распределяются между процессорами.

Заметьте: здесь предполагается, что потоки внутри процесса выполняют равные объемы работы. Но в многопоточном процессе это обычно не так; в нем есть, как правило, один или более «служебных» потоков (housekeeping threads) и несколько рабочих. Поэтому, если в многопоточном приложении нужно задействовать все преимущества многопроцессорной платформы, целесообразно указывать номера идеальных процессоров для потоков вызовом функции *SetThreadIdealProcessor*.

В системах с Hyperthreading следующим идеальным процессором является первый логический процессор на следующем физическом. Например, в двухпроцессорной системе с Hyperthreading логических процессоров — 4; если для первого потока идеальным процессором назначен логический процессор 0, то для второго потока имело бы смысл назначить таковым логический процессор 2, для третьего — логический процессор 1, для четвертого — логический процессор 3 и т. д. Тогда потоки равномерно распределялись бы по физическим процессорам.

В NUMA-системах идеальный узел для процесса выбирается при его (процесса) создании. Первому процессу назначается узел 0, второму — 1 и т. д. Затем идеальные процессоры для потоков процесса выбираются из идеального узла. Идеальным процессором для первого потока в процессе назначается первый процессор в узле. По мере создания дополнительных потоков в процессе за ними закрепляется тот же идеальный узел; следующий процессор в этом узле становится идеальным для следующего потока и т. д.

Алгоритмы планирования потоков в многопроцессорных системах

Теперь, описав типы многопроцессорных систем, поддерживаемых Windows, а также привязку потоков к процессорам и выбор идеального процессора, мы готовы объяснить вам применение этой информации при определении того, какие потоки выполняются и на каких процессорах. При этом система принимает два базовых решения:

- выбор процессора для потока, который готов к выполнению;
- выбор потока для конкретного процессора.

Выбор процессора для потока при наличии простаивающих процессоров

Как только поток готов к выполнению, Windows сначала пытается подключить его к простаивающему процессору. Если таких процессоров несколько, предпочтение отдается сначала идеальному процессору для данного потока, затем предыдущему, а потом текущему (т. е. процессору, на котором работает код, отвечающий за планирование). В Windows 2000, если все эти процессоры заняты, выбирается первый простаивающий процессор, на котором может работать данный поток, для чего сканируется маска свободных процессоров в направлении убывания их номеров.

В Windows XP и Windows Server 2003 выбор простаивающего процессора не так прост. Во-первых, выделяются простаивающие процессоры из чис-

ла тех, на которых маска привязки разрешает выполнение данного потока. Если система имеет архитектуру NUMA и в узле, где находится идеальный процессор для потока, есть простаивающие процессоры, то список всех простаивающих процессоров уменьшается до этого набора. Если в результате такой операции в списке не останется простаивающих процессоров, список не сокращается. Затем, если в системе работают процессоры с технологией Hyperthreading и имеется физический процессор, все логические процессоры которого свободны, список простаивающих процессоров уменьшается до этого набора. И вновь, если в результате такой операции в списке не останется простаивающих процессоров, список не сокращается.

Если текущий процессор (тот, который пытается определить, что делать с потоком, готовым к выполнению) относится к набору оставшихся простаивающих процессоров, поток планируется к выполнению именно на этом процессоре. А если текущий процессор не входит в список оставшихся простаивающих процессоров, если это система с технологией Hyperthreading и если есть простаивающий логический процессор на физическом, который содержит идеальный процессор для данного потока, то список простаивающих процессоров ограничивается этим набором. В ином случае система проверяет, имеются ли простаивающие логические процессоры на физическом, который содержит предыдущий процессор потока. Если такой набор не пуст, список простаивающих процессоров уменьшается до этого набора.

Из оставшегося набора простаивающих процессоров исключаются все процессоры, находящиеся в состоянии сна. (Эта операция не выполняется, если в ее результате такой список опустел бы.) Наконец, поток подключается к процессору с наименьшим номером в оставшемся списке.

Независимо от версии Windows, как только процессор выбран, соответствующий поток переводится в состояние Standby, и PRCB простаивающего процессора обновляется так, чтобы указывать на этот поток. При выполнении на этом процессоре цикл простоя обнаруживает, что поток выбран и подключает его к процессору.

Выбор процессора для потока в отсутствие простаивающих процессоров

Если простаивающих процессоров нет в момент, когда готовый к выполнению поток переведен в состояние Standby, Windows проверяет приоритет выполняемого потока (или того, который находится в состоянии Standby) и идеальный процессор для него, чтобы решить, следует ли вытеснить выполняемый. В Windows 2000 маска привязки может исключить идеальный процессор. (В Windows XP такое не допускается.) Если этот процессор не входит в маску привязки потока, Windows выбирает для потока процессор с наибольшим номером.

Если для идеального процессора уже выбран поток, ожидающий в состоянии Standby выделения процессорного времени, и его приоритет ниже, чем потока, готовящегося к выполнению, последний вытесняет первый и становится следующим выполняемым на данном процессоре. Если к процессору уже подключен какой-то поток, Windows сравнивает приоритеты текущего

и нового потока. Если приоритет текущего меньше, чем нового, первый помечается как подлежащий вытеснению, и Windows ставит в очередь межпроцессорное прерывание, чтобы целевой процессор вытеснил текущий поток в пользу нового.

> **ПРИМЕЧАНИЕ** Windows сравнивает приоритеты текущего и следующего потоков не на всех процессорах, а только на одном, выбранном по только что описанным правилам. Если вытеснение подключенного к данному процессору потока невозможно, новый поток помещается в очередь готовых потоков, соответствующую его уровню приоритета, где он и ждет выделения процессорного времени. Поэтому Windows не гарантирует первоочередное выполнение всех потоков с наивысшим приоритетом, но всегда выполняет один поток с таким приоритетом.

Как мы уже сказали, если готовый поток нельзя выполнить немедленно, он помещается в очередь готовых потоков и ждет выделения процессорного времени. Однако в Windows Server 2003 потоки всегда помещаются в очереди готовых потоков на своих идеальных процессорах.

Выбор потока для выполнения на конкретном процессоре (Windows 2000 и Windows XP)

В некоторых случаях (например, когда поток входит в состояние ожидания, снижает свой приоритет, изменяет привязку, откладывает выполнение или передает управление) Windows нужно найти новый поток для подключения к процессору, на котором работал текущий поток. Как уже говорилось, в однопроцессорной системе Windows просто выбирает первый поток из непустой очереди готовых потоков с наивысшим приоритетом. Но в многопроцессорной системе Windows 2000 или Windows XP должно быть соблюдено одно из дополнительных условий:

- поток уже выполнялся в прошлый раз на данном процессоре;
- данный процессор должен быть идеальным для этого потока;
- поток провел в состоянии Ready более трех тактов системного таймера;
- поток имеет приоритет не менее 24.

Таким образом, потоки с жесткой привязкой, в маску которых данный процессор не входит, очевидно, пропускаются. Если потоков, отвечающих одному из условий, нет, Windows отберет поток из начала той очереди готовых потоков, с которой она начинает поиск.

Почему так важно подключить поток именно к тому процессору, на котором он выполнялся в прошлый раз? Как обычно, все дело в быстродействии — процессор, на котором поток выполнялся в прошлый раз, скорее всего еще хранит его данные в своем кэше второго уровня.

Выбор потока для выполнения на конкретном процессоре (Windows Server 2003)

Поскольку в Windows Server 2003 у каждого процессора собственный список потоков, ждущих выполнения на этом процессоре, то по окончании выполнения текущего потока процессор просто проверяет свою очередь готовых потоков. Если его очереди пусты, к процессору подключается поток простоя. Затем этот поток начинает сканировать очереди готовых потоков при других процессорах и ищет потоки, которые можно было бы выполнять на данном процессоре. Заметьте, что в NUMA-системах поток простоя проверяет процессоры сначала в своем узле, а потом в других узлах.

Объекты-задания

Объект «задание» (job object) — это именуемый, защищаемый и разделяемый объект ядра, обеспечивающий управление одним или несколькими процессами как группой. Процесс может входить только в одно задание. По умолчанию его связь с объектом «задание» нельзя разрушить, и все процессы, создаваемые данным процессом и его потомками, будут сопоставлены с тем же заданием. Объект «задание» также регистрирует базовую учетную информацию всех включенных в него процессов, в том числе уже завершившихся. Windows-функции, предназначенные для создания объектов-заданий и манипулирования ими, перечислены в таблице 6-20.

Таблица 6-20. *Функции Windows API для заданий*

Функция	Описание
CreateJobObject	Создает объект-задание (с необязательным именем)
OpenJobObject	Открывает существующий объект-задание по имени
AssignProcessToJobObject	Включает процесс в задание
TerminateJobObject	Завершает все процессы в задании
SetInformationJobObject	Устанавливает ограничения на процессы задания
QueryInformationJobObject	Возвращает информацию о задании — количество использованного процессорного времени, счетчик числа ошибок страниц, число процессов, список идентификаторов процессов, квоты и лимиты защиты

Ниже кратко поясняются некоторые ограничения, которые можно налагать на задания.

- **Максимальное число активных процессов** Ограничивает число одновременно выполняемых процессов задания.
- **Общий лимит на процессорное время в пользовательском режиме** Ограничивает максимальное количество процессорного времени, потребляемого процессами задания (с учетом завершившихся) в пользовательском режиме. Как только этот лимит будет исчерпан, все процессы задания завершатся с сообщением об ошибке, а создание новых процессов в задании станет невозможным (если лимит не будет переустанов-

лен). Объект-задание будет переведен в свободное состояние, и все ожидавшие его потоки освободятся. Это поведение системы по умолчанию можно изменить через функцию *EndOfJobTimeAction*.

- **Индивидуальный лимит на процессорное время пользовательского режима для каждого процесса** Ограничивает максимальное количество процессорного времени, потребляемого каждым процессом в задании. По достижении этого лимита процесс завершается (не получая шанса на очистку).
- **Класс планирования задания** Устанавливает длительность кванта для потоков процессов, входящих в задание. Этот параметр применим только в системах, использующих длинные фиксированные кванты (в системах Windows Server по умолчанию). Длительность кванта определяется классом планирования задания, как показано в следующей таблице.

Класс планирования	Число квантовых единиц
0	6
1	12
2	18
3	24
4	30
5	36
6	42
7	48
8	54
9	Бесконечное для приоритета реального времени; в остальных случаях — 60

- **Привязка задания к процессорам** Устанавливает маску привязки к процессорам для каждого процесса задания. (Отдельные потоки могут изменять свои привязки на любое подмножество привязок задания, но процессы этого делать не могут.)
- **Класс приоритета для всех процессов задания** Определяет класс приоритета для каждого процесса в задании. Потоки не могут повышать свой приоритет относительно класса (как они это обычно делают). Все попытки повышения приоритета игнорируются. (При вызове *SetThreadPriority* ошибка не генерируется, но и приоритет не повышается.)
- **Минимальный и максимальный размеры рабочего набора по умолчанию** Устанавливает указанные минимальный и максимальный размеры рабочего набора для каждого процесса задания. (У каждого процесса свой рабочий набор, но с одинаковыми максимальным и минимальным размерами.)
- **Лимит на виртуальную память, передаваемую процессу или заданию** Указывает максимальный размер виртуального адресного пространства, который можно передать либо одному процессу, либо всему заданию.

Задания можно настроить на отправку в очередь объекта «порт завершения ввода-вывода» какого-либо элемента, который могут ждать другие потоки через Windows-функцию *GetQueuedCompletionStatus*.

Задание также позволяет накладывать на включенные в него процессы ограничения, связанные с защитой. Например, вы можете сделать так, чтобы все процессы в задании использовали один и тот же маркер доступа или не имели права олицетворять (подменять) другие процессы либо создавать процессы с маркерами доступа, включающими привилегии группы локальных администраторов. Кроме того, допускается применение фильтров защиты, предназначенных, например, для следующих ситуаций: когда потоки процессов задания олицетворяют клиентские потоки, из их маркера олицетворения можно избирательно исключать некоторые привилегии и идентификаторы защиты (SID).

Наконец, вы можете задавать ограничения для пользовательского интерфейса процессов задания, например запрещать открытие процессами описателей окон, которыми владеют потоки, не входящие в это задание, ограничивать операции с буфером обмена или блокировать изменение многих параметров пользовательского интерфейса системы с помощью Windows-функции *SystemParametersInfo*.

В Windows 2000 Datacenter Server имеется утилита Process Control Manager, позволяющая администратору определять объекты «задание», устанавливать для них различные квоты и лимиты, а также указывать процессы, которые следует включать в то или иное задание при запуске. Заметьте, что эта утилита больше не поставляется с Windows Server 2003 Datacenter Edition, но останется в системе при обновлении Windows 2000 Datacenter Server до Windows Server 2003 Datacenter Edition.

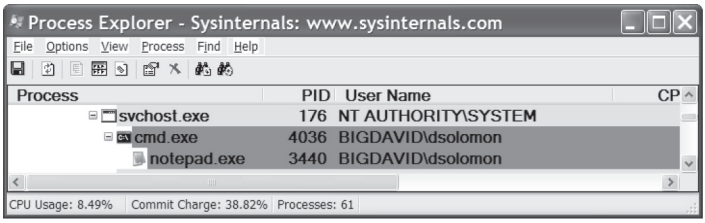
ЭКСПЕРИМЕНТ: просмотр объекта «задание»

Вы можете просматривать именованные объекты «задание» в оснастке Performance (Производительность). Для просмотра неименованных заданий нужно использовать команду *!job* или *dt nt!_ejob* отладчика ядра.

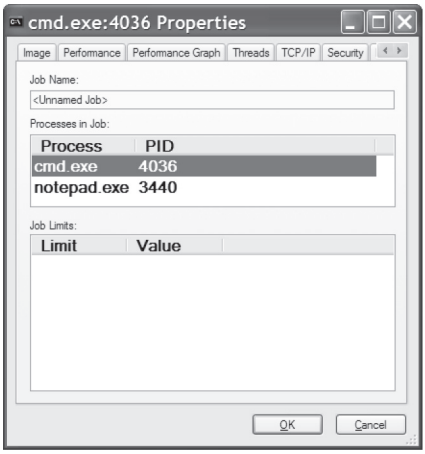
Выяснить, сопоставлен ли данный процесс с заданием, позволяет команда *!process* отладчика ядра или — в Windows XP и Windows Server 2003 — утилита Process Explorer. Чтобы создать неименованный объект «задание» и понаблюдать за ним, придерживайтесь следующей схемы.

1. Введите команду *runas* для создания процесса командной строки (Cmd.exe). Например, наберите runas /**user:<*домен*>\<*имя_пользователя*> cmd**. Далее введите свой пароль, и на экране появится окно командной строки. Windows-сервис, выполняющий команду *runas*, создаст неименованное задание, включающее все процессы (они будут завершены в момент вашего выхода из системы).
2. Из командной строки запустите Notepad.exe.
3. Запустите Process Explorer и обратите внимание на то, что процессы Cmd.exe и Notepad.exe выделяются как часть задания. Эти два процесса показаны на следующей иллюстрации.

см. след. стр.



4. Дважды щелкните либо процесс Cmd.exe, либо процесс Notepad.exe, чтобы открыть окно свойств. В этом окне вы увидите вкладку Job.
5. Перейдите на вкладку Job для просмотра детальных сведений о задании. В нашем случае с заданием не сопоставлены никакие квоты — в него просто включены два процесса.



6. Теперь запустите отладчик ядра в работающей системе (либо WinDbg в режиме локальной отладки ядра, либо LiveKd, если вы используете Windows 2000), выведите на экран список процессов командой *!process* и найдите в нем только что созданный процесс Cmd.exe. Затем просмотрите содержимое блока процесса, введя команду **!process <*идентификатор_процесса*>**, и найдите адрес объекта «задание». Наконец, исследуйте объект «задание» с помощью команды *!job*. Ниже приведен фрагмент вывода отладчика для этих команд в работающей системе:

```
lkd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
    ...
PROCESS 8567b758  SessionId: 0  Cid: 0fc4    Peb: 7ffdf000
    ParentCid: 00b0    DirBase: 1b3fb000  ObjectTable: e18dd7d0
    HandleCount:  19.    Image: cmd.exe

PROCESS 856561a0  SessionId: 0  Cid: 0d70    Peb: 7ffdf000
    ParentCid: 0fc4    DirBase: 2e341000  ObjectTable: e19437c8
```

```
    HandleCount:  16.    Image: notepad.exe

lkd> !process 0fc4
Searching for Process with Cid == fc4
PROCESS 8567b758  SessionId: 0  Cid: 0fc4    Peb: 7ffdf000
    ParentCid: 00b0    DirBase: 1b3fb000  ObjectTable: e18dd7d0
    HandleCount:  19.    Image: cmd.exe

    BasePriority                      8
    ...
    Job                               85557988

lkd> !job 85557988
Job at 85557988
  TotalPageFaultCount      0
  TotalProcesses           2
  ActiveProcesses          2
  TotalTerminatedProcesses 0
  LimitFlags               0
  MinimumWorkingSetSize    0
  MaximumWorkingSetSize    0
  ActiveProcessLimit       0
  PriorityClass            0
  UIRestrictionsClass      0
  SecurityLimitFlags       0
  Token                    00000000
```

7. Наконец, используйте команду *dt* для просмотра объекта-задания и обратите внимание на дополнительные поля:

```
lkd> dt nt!_ejob 85557988
nt!_EJOB
   +0x000 Event            : _KEVENT
   +0x010 JobLinks         : _LIST_ENTRY [ 0x805455c8 - 0x85797888 ]
   +0x018 ProcessListHead  : _LIST_ENTRY [ 0x8567b8dc - 0x85656324 ]
   +0x020 JobLock          : _ERESOURCE
   +0x058 TotalUserTime    : _LARGE_INTEGER 0x0
   +0x060 TotalKernelTime  : _LARGE_INTEGER 0x0
   +0x068 ThisPeriodTotalUserTime : _LARGE_INTEGER 0x0
   +0x070 ThisPeriodTotalKernelTime : _LARGE_INTEGER 0x0
   +0x078 TotalPageFaultCount : 0
   +0x07c TotalProcesses   : 2
   +0x080 ActiveProcesses  : 2
   +0x084 TotalTerminatedProcesses : 0
   +0x088 PerProcessUserTimeLimit : _LARGE_INTEGER 0x0
   +0x090 PerJobUserTimeLimit : _LARGE_INTEGER 0x0
   +0x098 LimitFlags       : 0
   +0x09c MinimumWorkingSetSize : 0
   +0x0a0 MaximumWorkingSetSize : 0
```

см. след. стр.

```
+0x0a4 ActiveProcessLimit : 0
+0x0a8 Affinity         : 0
+0x0ac PriorityClass    : 0 ''
+0x0b0 UIRestrictionsClass : 0
+0x0b4 SecurityLimitFlags : 0
+0x0b8 Token            : (null)
+0x0bc Filter           : (null)
+0x0c0 EndOfJobTimeAction : 0
+0x0c4 CompletionPort   : 0x8619d8c0
+0x0c8 CompletionKey    : (null)
+0x0cc SessionId        : 0
+0x0d0 SchedulingClass  : 5
+0x0d8 ReadOperationCount : 0
+0x0e0 WriteOperationCount : 0
+0x0e8 OtherOperationCount : 0
+0x0f0 ReadTransferCount : 0
+0x0f8 WriteTransferCount : 0
+0x100 OtherTransferCount : 0
+0x108 IoInfo           : _IO_COUNTERS
+0x138 ProcessMemoryLimit : 0
+0x13c JobMemoryLimit   : 0
+0x140 PeakProcessMemoryUsed : 0x256
+0x144 PeakJobMemoryUsed : 0x1f6
+0x148 CurrentJobMemoryUsed : 0x1f6
+0x14c MemoryLimitsLock : _FAST_MUTEX
+0x16c JobSetLinks      : _LIST_ENTRY [ 0x85557af4 - 0x85557af4 ]
+0x174 MemberLevel      : 0    +0x178 JobFlags         : 0
```

Резюме

Мы изучили структуру процессов, потоков и заданий, узнали, как они создаются, а также познакомились с алгоритмами распределения процессорного времени в Windows.

В этой главе было много ссылок на материалы, связанные с управлением памятью. Поскольку потоки выполняются в адресном пространстве процессов, следующим предметом рассмотрения станет управление виртуальной и физической памятью в Windows. Этому и посвящена глава 7.

ГЛАВА 7

Управление памятью

В этой главе вы узнаете, как реализована виртуальная память в Microsoft Windows и как осуществляется управление той частью виртуальной памяти, которая находится в физической. Мы также опишем внутреннюю структуру диспетчера памяти и его компоненты, в том числе ключевые структуры данных и алгоритмы. Прежде чем изучать механизмы управления памятью, давайте рассмотрим базовые сервисы, предоставляемые диспетчером памяти, и основные концепции, такие как зарезервированная (reserved memory), переданная (committed memory) и разделяемая память (shared memory).

Введение в диспетчер памяти

По умолчанию виртуальный размер процесса в 32-разрядной Windows — 2 Гб. Если образ помечен как поддерживающий большое адресное пространство и система загружается со специальным ключом (о нем мы расскажем позже), 32-разрядный процесс может занимать до 3 Гб в 32-разрядной Windows и до 4 Гб в 64-разрядной. Размер виртуального адресного пространства процесса в 64-разрядной Windows составляет 7152 Гб на платформе IA64 и 8192 Гб на платформе x64. (Это значение может увеличиться в следующих выпусках 64-разрядной Windows.)

Как вы видели в главе 2 (особенно в таблице 2-4), максимальный объем физической памяти, поддерживаемый Windows, варьируется от 2 до 1024 Гб в зависимости от версии и редакции Windows. Так как виртуальное адресное пространство может быть больше или меньше объема физической памяти в компьютере, диспетчер управления памятью решает две главные задачи.

- Трансляция, или проецирование (mapping), виртуального адресного пространства процесса на физическую память. Это позволяет ссылаться на корректные адреса физической памяти, когда потоки, выполняемые в контексте процесса, читают и записывают в его виртуальном адресном пространстве. Физически резидентное подмножество виртуального адресного пространства процесса называется *рабочим набором* (working set).
- Подкачка части содержимого памяти на диск, когда потоки или системный код пытаются задействовать больший объем физической памяти, чем тот, который имеется в наличии, и загрузка страниц обратно в физическую память по мере необходимости.

Кроме управления виртуальной памятью диспетчер памяти предоставляет базовый набор сервисов, на которые опираются различные подсистемы окружения Windows. К этим сервисам относится поддержка файлов, проецируемых в память (memory-mapped files) [их внутреннее название — *объекты-разделы* (section objects)], памяти, копируемой при записи, и приложений, использующих большие разреженные адресные пространства. Диспетчер памяти также позволяет процессу выделять и использовать большие объемы физической памяти, чем можно спроецировать на виртуальное адресное пространство процесса (например, в 32-разрядных системах, в которых установлено более 4 Гб физической памяти). Соответствующий механизм поясняется в разделе «Address Windowing Extensions» далее в этой главе.

Компоненты диспетчера памяти

Диспетчер памяти является частью исполнительной системы Windows, содержится в файле Ntoskrnl.exe и включает следующие компоненты.

- Набор сервисов исполнительной системы для выделения, освобождения и управления виртуальной памятью; большинство этих сервисов доступно через Windows API или интерфейсы драйверов устройств режима ядра.
- Обработчики ловушек трансляции недействительных адресов (translation-not-valid) и нарушений доступа для разрешения аппаратно обнаруживаемых исключений, связанных с управлением памятью, а также загрузки в физическую память необходимых процессу страниц.
- Несколько ключевых компонентов, работающих в контексте шести различных системных потоков режима ядра.
  - *Диспетчер рабочих наборов* (working set manager) с приоритетом 16. Диспетчер настройки баланса (системный поток, создаваемый ядром) вызывает его раз в секунду или при уменьшении объема свободной памяти ниже определенного порогового значения. Он реализует общие правила управления памятью, например усечение рабочего набора, старение и запись модифицированных страниц.
  - *Поток загрузки и выгрузки стеков* (process/stack swapper) с приоритетом 23. Выгружает (outswapping) и загружает (inswapping) стеки процесса и потока. При необходимости операций со страничным файлом этот поток пробуждается диспетчером рабочих наборов и кодом ядра, отвечающим за планирование.
  - *Подсистема записи модифицированных страниц* (modified page writer) с приоритетом 17. Записывает измененные страницы, зарегистрированные в списке модифицированных страниц, обратно в соответствующие страничные файлы. Этот поток пробуждается, когда возникает необходимость в уменьшении размера списка модифицированных страниц.
  - *Подсистема записи спроецированных страниц* (mapped page writer) с приоритетом 17. Записывает измененные страницы спроецированных файлов на диск. Пробуждается, когда нужно уменьшить размер

списка модифицированных страниц или когда страницы модифицированных файлов находятся в этом списке более 5 минут. Этот второй поток записи модифицированных страниц требуется потому, что он может генерировать ошибки страниц, в результате которых выдаются запросы на свободные страницы. Если бы в системе был лишь один поток записи модифицированных страниц, она могла бы перейти в бесконечное ожидание свободных страниц.

- *Поток сегмента разыменования* (dereference segment thread) с приоритетом 18. Отвечает за уменьшение размеров системного кэша и изменение размеров страничного файла.
- *Поток обнуления страниц* (zero page thread) с приоритетом 0. Заполняет нулями страницы, зарегистрированные в списке свободных страниц. (В некоторых случаях обнуление памяти выполняется более скоростной функцией *MiZeroInParallel*.)

Внутренняя синхронизация

Как и другие компоненты исполнительной системы Windows, диспетчер памяти полностью реентерабелен и поддерживает одновременное выполнение в многопроцессорных системах, управляя тем, как потоки захватывают ресурсы. С этой целью диспетчер памяти контролирует доступ к собственным структурам данным, используя внутренние механизмы синхронизации, например спин-блокировку и ресурсы исполнительной системы (о синхронизирующих объектах см. главу 3).

Диспетчер памяти должен синхронизировать доступ к таким общесистемным ресурсам, как база данных номеров фреймов страниц (PFN) (контроль через спин-блокировку), объекты «раздел» и системный рабочий набор (контроль через спин-блокировку с заталкиванием указателя) и страничные файлы (контроль через объекты «мьютекс»). В Windows XP и Windows Server 2003 ряд таких блокировок был либо удален, либо оптимизирован, что позволило резко снизить вероятность конкуренции. Например, в Windows 2000 для синхронизации изменений в системном адресном пространстве и при передаче памяти применялись спин-блокировки, но, начиная с Windows XP, эти спин-блокировки были удалены, чтобы повысить масштабируемость. Индивидуальные для каждого процесса структуры данных управления памятью, требующие синхронизации, включают блокировку рабочего набора (удерживаемую на время внесения изменений в список рабочего набора) и блокировку адресного пространства (удерживаемую в период его изменения). Синхронизация рабочего набора в Windows 2000 реализована с помощью мьютекса, но в Windows XP и более поздних версиях применяется более эффективная блокировка с заталкиванием указателя, которая поддерживает как разделяемый, так и монопольный доступ.

К другим операциям, в которых больше не используется захват блокировок, относятся контроль квот на пулы подкачиваемой и неподкачиваемой памяти, управление передачей страниц, а также выделение и проецирование

физической памяти через функции поддержки AWE (Address Windowing Extensions). Кроме того, блокировка, синхронизирующая доступ к структурам, которые описывают физическую память (база данных PFN), теперь захватывается реже и удерживается в течение меньшего времени. Эти изменения особенно важны в многопроцессорных системах, где они позволили уменьшить частоту блокировки диспетчера памяти на период модификации со стороны другого процессора какой-либо глобальной структуры или вообще исключить такую блокировку.

Конфигурирование диспетчера памяти

Как и большинство компонентов Windows, диспетчер памяти старается автоматически оптимизировать работу систем различных масштабов и конфигураций при разных уровнях загруженности. Некоторые стандартные настройки можно изменить через параметры в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management, но, как правило, они оптимальны в большинстве случаев.

Многие пороговые значения и лимиты, от которых зависит политика принятия решений диспетчером памяти, вычисляются в период загрузки системы на основе доступной памяти и типа продукта (Windows 2000 Professional, Windows XP Professional и Windows XP Home Edition оптимизируется для интерактивного использования в качестве персональной системы, а системы Windows Server — для поддержки серверных приложений). Эти значения записываются в различные переменные ядра и впоследствии используются диспетчером памяти. Некоторые из них можно найти поиском в Ntoskrnl.exe глобальных переменных с именами, которые начинаются с *Mm* и содержат слово «maximum» или «minimum».

> **ВНИМАНИЕ** Не изменяйте значения этих переменных. Как показывают результаты тестирования, автоматически вычисляемые значения обеспечивают оптимальное быстродействие. Их модификация может привести к непредсказуемым последствиям вплоть до зависания и даже краха.

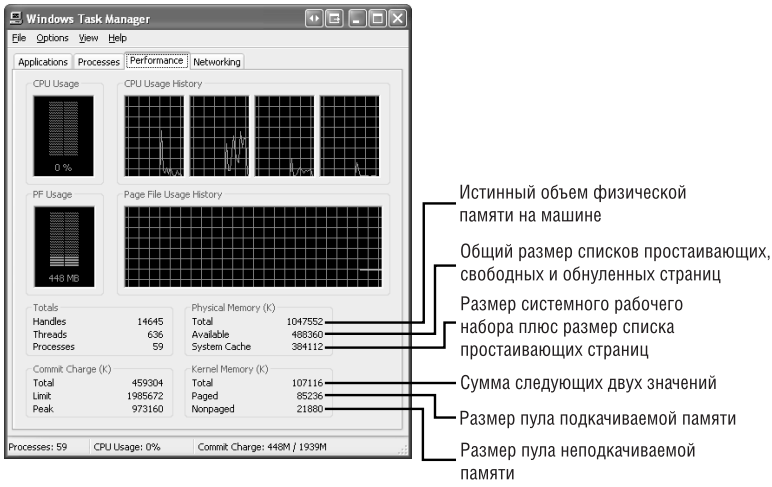
Исследование используемой памяти

Объекты счетчиков производительности Memory (Память) и Process (Процесс) открывают доступ к большей части сведений об использовании памяти системой и процессами. В этой главе мы нередко упоминаем счетчики, относящиеся к рассматриваемым компонентам.

Кроме оснастки Performance (Производительность) информацию об использовании памяти выводят некоторые утилиты из Windows Support Tools и ресурсов Windows. Мы включили ряд примеров и экспериментов, иллюстрирующих их применение. Но предупреждаем: одна и та же информация по-разному называется в разных утилитах. Это демонстрирует следующий эксперимент (определения упоминаемых в нем терминов будут даны в других разделах).

ЭКСПЕРИМЕНТ: просмотр информации о системной памяти

Базовую информацию о системной памяти можно получить на вкладке Performance (Быстродействие) в Task Manager (Диспетчер задач), как показано ниже (здесь используется Windows XP). Эти сведения являются подмножеством информации о памяти, предоставляемой счетчиками производительности.



Как Pmon.exe (из Windows Support Tools), так и Pstat.exe (из Platform SDK) выводят сведения о памяти системы и процессов. Взгляните на образец вывода Pstat (определения некоторых терминов см. в таблице 7-15).

Общий объем физической памяти

Общий размер списков простаивающих, свободных и обнуленных страниц

Суммарный размер рабочих наборов процессов (он не соответствует общему объему используемой процессами памяти, так как число разделяемых страниц удваивается)

Резидентный системный код

Резидентная часть пула подкачиваемой памяти

```
Memory:  64692K     Avail: 3676K     TotalWs:  62836K     InRam Kernel: 3788K     P: 9604K
Commit: 111848K/85808K     Limit: 182400K     Peak: 113732K     Pool N: 2344K     P:14448K
```

Размер физической части пула неподкачиваемой памяти

Размер виртуальной части пула подкачиваемой памяти

Размер системного рабочего набора (не только размер файлового кэша)

User Time	Kernel Time	Ws 20032	Faults 194378	Commit	Pri	Hnd	Thd	Pid	Name File Cache
0:00:00.000	8:03:55.734	16	1	0	0	0	2	0	Idle Process
0:00:00.000	0:01:24.421	32	1601	36	8	140	27	2	System
0:00:00.140	0:00:00.468	200	2008	164	11	35	6	20	smss.exe
0:00:00.640	0:00:02.843	716	8625	1588	13	145	9	30	csrss.exe

см. след. стр.

Для просмотра использованного объема памяти подкачиваемого и неподкачиваемого пулов по отдельности используйте утилиту Poolmon, описанную в разделе «Мониторинг использования пулов».

Наконец, команда *!vm* отладчика ядра выводит базовые сведения об управлении памятью, доступные через соответствующие счетчики производительности. Эта команда может быть полезна при изучении аварийного дампа или зависшей системы. Пример вывода этой команды приводится ниже.

```
kd> !vm

*** Virtual Memory Usage ***
        Physical Memory:    32620   (  130480 Kb)
        Page File: \??\C:\pagefile.sys
           Current:    204800Kb Free Space:    101052Kb
           Minimum:    204800Kb Maximum:       204800Kb
        Available Pages:     3604   (   14416 Kb)
        ResAvail Pages:     24004   (   96016 Kb)
        Modified Pages:       768   (    3072 Kb)
        NonPagedPool Usage:  1436   (    5744 Kb)
        NonPagedPool Max:   12940   (   51760 Kb)
        PagedPool 0 Usage:   6817   (   27268 Kb)
        PagedPool 1 Usage:    982   (    3928 Kb)
        PagedPool 2 Usage:    984   (    3936 Kb)
        PagedPool Usage:     8783   (   35132 Kb)
        PagedPool Maximum:  26624   (  106496 Kb)
        Shared Commit:       1361   (    5444 Kb)
        Special Pool:           0   (       0 Kb)
        Free System PTEs:  189291   (  757164 Kb)
        Shared Process:      3165   (   12660 Kb)
        PagedPool Commit:    8783   (   35132 Kb)
        Driver Commit:       1098   (    4392 Kb)
        Committed pages:    45113   (  180452 Kb)
        Commit limit:       79556   (  318224 Kb)

        Total Private:      30536   (  122144 Kb)
            IEXPLORE.EXE       3028 (   12112 Kb)
            svchost.exe        2128 (    8512 Kb)
            WINWORD.EXE        1971 (    7884 Kb)
            POWERPNT.EXE       1905 (    7620 Kb)
            Acrobat.exe        1761 (    7044 Kb)
            winlogon.exe       1361 (    5444 Kb)
            explorer.exe       1300 (    5200 Kb)
            livekd.exe         1015 (    4060 Kb)
            hh.exe              960 (    3840 Kb)
            ...
```

ЭКСПЕРИМЕНТ: учет использованной физической памяти

Комбинируя данные от счетчиков производительности и выходную информацию команд отладчика ядра, можно получить довольно полное представление об использовании физической памяти компьютера под управлением Windows. Соответствующие счетчики производительности доступны через оснастку Performance. (Вам будет удобнее, если вы установите максимальное значение для вертикальной шкалы равным 1000.)

- **Суммарный размер рабочих наборов процессов** Для просмотра этих данных выберите объект Process (Процесс) и счетчик Working Set (Рабочее множество) для экземпляра _Total. Показываемое значение превышает реальный объем используемой памяти, так как разделяемые страницы учитываются в каждом рабочем наборе процесса, использующего эти страницы. Более точную картину использования памяти процессами вы получите, вычтя из общего объема физической памяти следующие показатели: размер свободной памяти, объем памяти, занятой операционной системой (неподкачиваемый пул, резидентная часть подкачиваемого пула и резидентный код операционной системы и драйверов), а также размер списка модифицированных страниц. Оставшаяся часть соответствует памяти, используемой процессами. Сравнив ее размер с размером рабочего набора процесса, показываемым оснасткой Performance, можно получить намек на объем разделяемой памяти. Хотя исследование использованной физической памяти — дело весьма увлекательное, гораздо важнее знать, сколько закрытой виртуальной памяти передано процессам, — утечки памяти проявляются как увеличение объема закрытой виртуальной памяти, а не размера рабочего набора. На каком-то этапе диспетчер памяти остановит чрезмерные аппетиты процесса, но размер виртуальной памяти может расти, пока не достигнет общесистемного лимита (максимально возможного в данной системе объема закрытой переданной памяти) либо лимита, установленного для задания или процесса (если процесс включен в задание); см. раздел «Страничные файлы» далее в этой главе.
- **Суммарный размер системного рабочего набора** Эти данные можно увидеть, выбрав объект Memory (Память) и счетчик Cache Bytes (Байт кэш-памяти). Как поясняется в разделе «Системный рабочий набор», суммарный размер системного рабочего набора определяется не только размером кэша, но и подмножеством пула подкачиваемой памяти и объемом резидентного кода операционной системы и драйверов, находящегося в этом рабочем наборе.
- **Размер пула неподкачиваемой памяти** Для просмотра этого значения выберите счетчик Memory: Pool Nonpaged Bytes (Память: Байт в невыгружаемом страничном пуле).

см. след. стр.

- **Размер списков простаивающих, свободных и обнуленных страниц** Общий размер этих списков сообщает счетчик Memory: Available Bytes (Память: Доступно байт). Если вы хотите узнать размер каждого из списков, используйте команду *!memusage* отладчика ядра.

Теперь на графике (или в отчете) должна присутствовать информация обо всей физической памяти, кроме двух элементов, о которых счетчики производительности не сообщают:

- неподкачиваемого кода операционной системы и драйверов;
- списка модифицированных страниц и списка модифицированных, но не записываемых страниц (modified-no-write paging list).

Хотя размеры двух последних списков легко узнать с помощью команды *!memusage* отладчика ядра, выяснить размер резидентного кода операционной системы и драйверов не так просто.

Сервисы диспетчера памяти

Диспетчер памяти предоставляет набор системных сервисов для выделения и освобождения виртуальной памяти, разделения памяти между процессами, проецирования файлов в память, сброса виртуальных страниц на диск, получения информации о диапазоне виртуальных страниц, изменения атрибутов защиты виртуальных страниц и блокировки в памяти.

Как и другие сервисы исполнительной системы, сервисы управления памятью требуют при вызове передачи описателя того процесса, с виртуальной памятью которого будут проводиться операции. Таким образом, вызывающая программа может управлять как собственной памятью, так и памятью других процессов (при наличии соответствующих прав). Например, если один процесс порождает другой, у первого по умолчанию остается право на манипуляции с виртуальной памятью второго. Впоследствии родительский процесс может выделять и освобождать память, считывать и записывать в нее данные через сервисы управления виртуальной памятью, передавая им в качестве аргумента описатель дочернего процесса. Подсистемы используют эту возможность для управления памятью своих клиентских процессов; она же является ключевой для реализации отладчиков, так как им нужен доступ к памяти отлаживаемого процесса для чтения и записи.

Большинство этих сервисов предоставляется через Windows API. В него входят три группы прикладных функций управления памятью: для операций со страницами виртуальной памяти (*Virtualxxx*), проецирования файлов в память (*CreateFileMapping*, *MapViewOfFile*) и управления кучами (*Heapxxx*, а также функции из старых версий интерфейса — *Localxxx* и *Globalxxx*).

Диспетчер памяти поддерживает такие сервисы, как выделение и освобождение физической памяти, блокировка страниц в физической памяти для передачи данных другим компонентам исполнительной системы режима ядра и драйверам устройств через DMA. Имена этих функций начинают-

ся с префикса *Mm*. Кроме того, существуют процедуры поддержки исполнительной системы, имена которых начинаются с *Ex*. Не являясь частью диспетчера памяти в строгом смысле этого слова, они применяются для выделения и освобождения памяти из системных куч (пулов подкачиваемой и неподкачиваемой памяти), а также для манипуляций с ассоциативными списками. Мы затронем эту тематику в разделе «Системные пулы памяти» далее в этой главе.

Несмотря на упоминание Windows-функций управления памятью в режиме ядра и выделения памяти для драйверов устройств, мы будем рассматривать не столько интерфейсы и особенности их программирования, сколько внутренние принципы работы этих функций. Полное описание доступных функций и их интерфейсов см. в документации Platform SDK и DDK.

Большие и малые страницы

Виртуальное адресное пространство делится на единицы, называемые страницами. Это вызвано тем, что аппаратный блок управления памятью транслирует виртуальные адреса в физические по страницам. Поэтому страница — наименьшая единица защиты на аппаратном уровне. (Различные параметры защиты страниц описываются в разделе «Защита памяти» далее в этой главе.) Страницы бывают двух размеров: малого и большого. Реальный размер зависит от аппаратной платформы (см. таблицу 7-1).

Таблица 7-1. *Размеры страниц*

Архитектура	Размер малой страницы	Размер большой страницы
x86	4 Кб	4 Мб (2 Мб в PAE-системах)
x64	4 Кб	2 Мб
IA64	8 Кб	16 Мб

Преимущество больших страниц — скорость трансляции адресов для ссылок на другие данные в большой странице. Дело в том, что первая ссылка на любой байт внутри большой страницы заставляет аппаратный ассоциативный буфер трансляции (translation look-aside buffer, TLB) (см. раздел «Ассоциативный буфер трансляции» далее в этой главе) загружать в свой кэш информацию, необходимую для трансляции ссылок на любые другие байты в этой большой странице. При использовании малых страниц для того же диапазона виртуальных адресов требуется больше элементов TLB, что заставляет чаще обновлять элементы по мере трансляции новых виртуальных адресов. А это в свою очередь требует чаще обращаться к структурам таблиц страниц при ссылках на виртуальные адреса, выходящие за пределы данной малой страницы. TLB — очень маленький кэш, и поэтому большие страницы обеспечивают более эффективное использование этого ограниченного ресурса.

Чтобы задействовать преимущества больших страниц в системах с достаточным объемом памяти (см. минимальные размеры памяти в таблице 7-2), Windows проецирует на такие страницы базовые образы операционной сис-

темы (Ntoskrnl.exe и Hal.dll) и базовые системные данные (например, начальную часть пула неподкачиваемой памяти и структуры данных, описывающие состояние каждой страницы физической памяти). Windows также автоматически проецирует на большие страницы запросы объемного ввода-вывода (драйверы устройств вызывают *MmMapIoSpace*), если запрос удовлетворяет длине и выравниванию для большой страницы. Наконец, Windows разрешает приложениям проецировать на такие страницы свои образы, закрытые области памяти и разделы, поддерживаемые страничным файлом (pagefile-backed sections). (См. описание флага MEM_LARGE_PAGE функции *VirtualAlloc*.) Вы можете указать, чтобы и другие драйверы устройств проецировались на большие страницы, добавив многострочный параметр реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\LargePageDrivers и задав имена драйверов как отдельные строки с нулем в конце.

Таблица 7-2. *Минимальные объемы физической памяти, при которых включается поддержка больших страниц*

Операционная система	Минимальная физическая память
Windows 2000	>127 Мб
Windows XP, Windows Server 2003	>255 Мб

Один из побочных эффектов применения больших страниц заключается в следующем. Так как аппаратная защита памяти оперирует страницами как наименьшей единицей, то, если на большой странице содержатся код только для чтения и данные для записи/чтения, она должна быть помечена как доступная для чтения и записи, т. е. код станет открытым для записи. А значит, драйверы устройств или другой код режима ядра мог бы в результате скрытой ошибки модифицировать код операционной системы или драйверов, изначально предполагавшийся только для чтения, и не вызвать нарушения доступа к памяти. Однако при использовании малых страниц для проецирования ядра части NTOSKRNL.EXE и HAL.DLL только для чтения будут спроецированы именно как страницы только для чтения. Хотя это снижает эффективность трансляции адресов, зато при попытке драйвера устройства (или другого кода режима ядра) модифицировать доступную только для чтения часть операционной системы произойдет немедленный крах с указанием на неверную инструкцию. Поэтому, если вы подозреваете, что источник ваших проблем связан с повреждением кода ядра, включите Driver Verifier — это автоматически отключит использование больших страниц.

Резервирование и передача страниц

Страницы в адресном пространстве процесса могут быть свободными (free), зарезервированными (reserved) или переданными (committed). Приложения могут *резервировать* (reserve) адресное пространство и *передавать память* (commit) зарезервированным страницам по мере необходимости. Резервировать страницы и передавать им память можно одним вызовом. Эти сервисы предоставляются через Windows-функции *VirtualAlloc* и *VirtualAllocEx*.

Резервирование адресного пространства позволяет потоку резервировать диапазон виртуальных адресов для последующего использования. Попытка доступа к зарезервированной памяти влечет за собой нарушение доступа, так как ее страницы не спроецированы на физическую память.

При попытке доступа адреса переданных страниц в конечном счете транслируются в допустимые адреса страниц физической памяти. Переданные страницы могут быть закрытыми (не предназначенными для разделения с другими процессами) или спроецированными на представление объекта-раздела (на которое в свою очередь могут проецировать страницы другие процессы).

Закрытые страницы процесса, к которым еще не было обращения, создаются при первой попытке доступа как обнуленные. Закрытые переданные страницы могут впоследствии записываться операционной системой в страничный файл (в зависимости от текущей ситуации). Такие страницы недоступны другим процессам, если только они не используют функции *ReadProcessMemory* или *WriteProcessMemory*. Если переданные страницы спроецированы на часть проецируемого файла, их скорее всего придется загрузить с диска — при условии, что они не были считаны раньше из-за обращения к ним того же или другого процесса, на который спроецирован этот файл.

Страницы записываются на диск по обычной процедуре записи модифицированных страниц, которые перемещаются из рабочего набора процесса в список модифицированных страниц и в конечном счете на диск (о рабочих наборах и списке модифицированных страниц — чуть позже). Страницы проецируемого файла можно сбросить на диск явным вызовом функции *FlushViewOfFile*.

Для возврата страниц (decommitting) и/или освобождения виртуальной памяти предназначена функция *VirtualFree* или *VirtualFreeEx*. Различия между возвратом и освобождением страниц такие же, как между резервированием и передачей: возвращенная память все еще зарезервирована, тогда как освобожденная память действительно свободна и не является ни переданной, ни зарезервированной.

Такой двухэтапный процесс (резервирование и передача) помогает снизить нагрузку на память, откладывая передачу страниц до реальной необходимости в них. Резервирование памяти — операция относительно быстрая и не требующая большого количества ресурсов, поскольку в данном случае не расходуется ни физическая память (драгоценный системный ресурс), ни квота процесса на ресурсы страничного файла (число страниц, передаваемых процессу из страничного файла). При этом нужно создать или обновить лишь сравнительно небольшие внутренние структуры данных, отражающие состояние адресного пространства процесса. (Об этих структурах данных, называемых дескрипторами виртуальных адресов, или VAD, мы расскажем потом.)

Резервирование памяти с последующей ее передачей особенно эффективно для приложений, нуждающихся в потенциально большой и непрерывной области виртуальной памяти: зарезервировав требуемое адресное пространство, они могут передавать ему страницы порциями, по мере необхо-

димости. Эта методика применяется и для организации стека пользовательского режима для каждого потока. Такой стек резервируется при создании потока. (Его размер по умолчанию — 1 Мб; другой размер стека для конкретного потока можно указать при вызове *CreateThread*. Если вы хотите изменить его для всех потоков процесса, укажите при сборке программы флаг /STACK.) По умолчанию стеку передается только начальная страница, а следующая страница просто помечается как *сторожевая* (guard page). За счет этой страницы, которая служит своего рода ловушкой для перехвата ссылок за ее пределы, стек расширяется только по мере заполнения.

Блокировка памяти

В целом, принятие решений о том, какие страницы следует оставить в физической памяти, лучше сохранить за диспетчером памяти. Однако в особых обстоятельствах можно подкорректировать работу диспетчера памяти. Существует два способа блокировки страниц в памяти.

- Windows-приложения могут блокировать страницы в рабочем наборе своего процесса через функцию *VirtualLock*. Максимальное число страниц, которые процесс может блокировать, равно минимальному размеру его рабочего набора за вычетом восьми страниц. Следовательно, если процессу нужно блокировать большее число страниц, он может увеличить минимальный размер своего рабочего набора вызовом функции *SetProcessWorkingSetSize* (см. раздел «Управление рабочим набором» далее в этой главе).
- Драйверы устройств могут вызывать функции режима ядра *MmProbeAndLockPages*, *MmLockPagableCodeSection* и *MmLockPagableSectionByHandle*. Блокированные страницы остаются в памяти до снятия блокировки. Хотя число блокируемых страниц не ограничивается, драйвер не может блокировать их больше, чем это позволяет счетчик доступных резидентных страниц.

Гранулярность выделения памяти

Windows выравнивает начало каждого региона зарезервированного адресного пространства в соответствии с *гранулярностью выделения памяти* (allocation granularity). Это значение можно получить через Windows-функцию *GetSystemInfo*. В настоящее время оно равно 64 Кб. Такая величина выбрана из соображений поддержки будущих процессоров с бóльшим размером страниц памяти (до 64 Кб) или виртуально индексируемых кэшей (virtually indexed caches), требующих общесистемного выравнивания между физическими и виртуальными страницами (physical-to-virtual page alignment). Благодаря этому уменьшается риск возможных изменений, которые придется вносить в приложения, полагающиеся на определенную гранулярность выделения памяти. (Это ограничение не относится к коду Windows режима ядра — используемая им гранулярность выделения памяти равна одной странице.)

Windows также добивается, чтобы размер и базовый адрес зарезервированного региона адресного пространства всегда был кратен размеру страницы. Например, системы типа x86 используют страницы размером 4 Кб, и, если вы попытаетесь зарезервировать 18 Кб памяти, на самом деле будет зарезервировано 20 Кб. А если вы укажете базовый адрес 3 Кб для 18-килобайтного региона, то на самом деле будет зарезервировано 24 Кб.

Разделяемая память и проецируемые файлы

Как и большинство современных операционных систем, Windows поддерживает механизм разделения памяти. *Разделяемой* (shared memory) называется память, видимая более чем одному процессу или присутствующая в виртуальном адресном пространстве более чем одного процесса. Например, если два процесса используют одну и ту же DLL, есть смысл загрузить ее код в физическую память лишь один раз и сделать ее доступной всем процессам, проецирующим эту DLL (рис. 7-1).

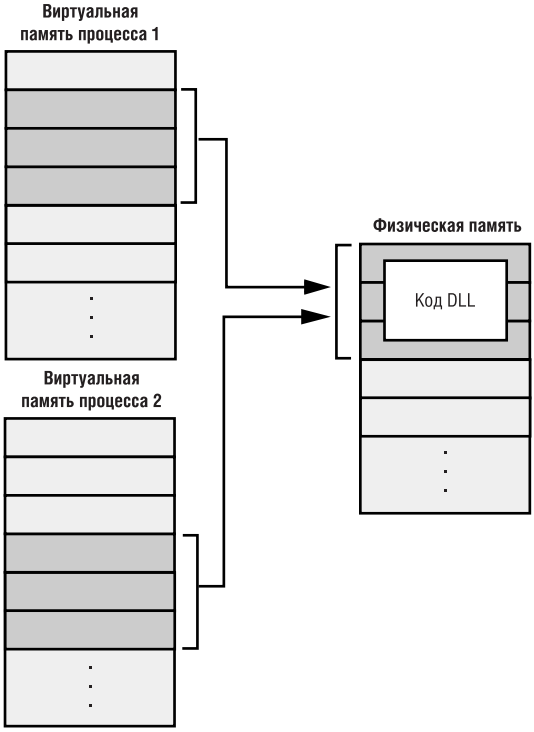


Рис. 7-1. *Разделение памяти процессами*

Каждый процесс поддерживает закрытые области памяти для хранения собственных данных, но программные инструкции и страницы немодифицируемых данных в принципе можно использовать совместно с другими процессами. Как вы еще увидите, такой вид разделения реализуется автома-

тически, поскольку страницы кода в исполняемых образах проецируются с атрибутом «только для выполнения», а страницы, доступные для записи, — с атрибутом «копирование при записи» (copy-on-write) (см. раздел «Копирование при записи» далее в этой главе).

Для реализации разделяемой памяти используются примитивы диспетчера памяти, объекты «раздел», которые в Windows API называются *объектами «проекция файла»* (file mapping objects). Внутренняя структура и реализация этих объектов описывается в разделе «Объекты-разделы» далее в этой главе.

Этот фундаментальный примитив диспетчера памяти применяется для проецирования виртуальных адресов в основной памяти, страничном файле или любых других файлах, к которым приложение хочет обращаться так, будто они находятся в памяти. Раздел может быть открыт как одним процессом, так и несколькими; иначе говоря, объекты «раздел» вовсе не обязательно представляют разделяемую память.

Объект «раздел» может быть связан с открытым файлом на диске (который в этом случае называется проецируемым) или с переданной памятью (для ее разделения). Разделы, проецируемые на переданную память, называются *разделами, поддерживаемыми страничными файлами* (page file backed sections), так как при нехватке памяти их страницы перемещаются в страничный файл. (Однако Windows может работать без страничного файла, и тогда эти разделы «поддерживаются» физической памятью.) Разделяемые переданные страницы, как и любые другие страницы, видимые в пользовательском режиме (например, закрытые переданные страницы), всегда обнуляются при первом обращении к ним.

Для создания объекта «раздел» используется Windows-функция *CreateFileMapping*, которой передается описатель проецируемого файла (или INVALID_ HANDLE_VALUE в случае раздела, поддерживаемого страничным файлом), а также необязательные имя и дескриптор защиты. Если разделу присвоено имя, его может открыть другой процесс вызовом *OpenFileMapping*. Кроме того, вы можете предоставить доступ к объектам «раздел» через наследование описателей (определив при открытии или создании описателя, что он является наследуемым) или их дублирование (с помощью *DuplicateHandle*). Драйверы также могут манипулировать объектами «раздел» через функции *ZwOpenSection*, *ZwMapViewOfSection* и *ZwUnmapViewOfSection*.

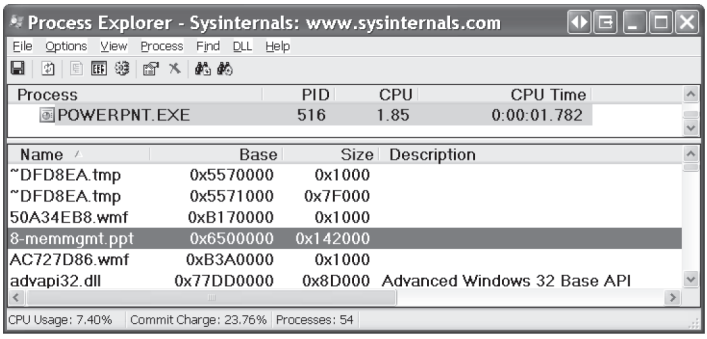
Объект «раздел» может ссылаться на файлы, длина которых намного превышает размер адресного пространства процесса. (Если раздел поддерживается страничным файлом, в нем должно быть достаточно места для размещения всего раздела.) Используя очень большой объект «раздел», процесс может проецировать лишь необходимую ему часть этого объекта, которая называется *представлением* (view) и создается вызовом функции *MapViewOfFile* с указанием проецируемого диапазона. Это позволяет процессам экономить адресное пространство, так как на память проецируется только представление объекта «раздел».

Windows-приложения могут использовать проецирование файлов для упрощения ввода-вывода в файлы на диске, просто делая их доступными в

своем адресном пространстве. Приложения — не единственные потребители объектов «раздел»: загрузчик образов использует их для проецирования в память исполняемых образов, DLL и драйверов устройств, а диспетчер кэша — для доступа к данным кэшируемых файлов. (Об интеграции диспетчера кэша с диспетчером памяти см. в главе 11.) О реализации разделов совместно используемой памяти мы расскажем потом.

ЭКСПЕРИМЕНТ: просмотр файлов, проецируемых в память

Просмотреть спроецированные в память файлы для какого-либо процесса позволяет утилита Process Explorer от Sysinternals. Для этого настройте нижнюю секцию ее окна на режим отображения DLL. (Выберите View, Lower Pane View, DLLs.) Заметьте, что это не просто список DLL, — здесь представлены все спроецированные в память файлы в адресном пространстве процесса. Некоторые являются DLL, один из них — файлом выполняемого образа (EXE), а другие элементы списка могут представлять файлы данных, проецируемые в память. Например, на следующей иллюстрации показан вывод Process Explorer применительно к процессу Microsoft PowerPoint, в адресное пространство которого загружен документ PowerPoint.



Для поиска спроецированных в память файлов щелкните Find, DLL. Это удобно, когда нужно определить, какие процессы используют DLL, которую вы пытаетесь заменить.

Наконец, сравнение списка DLL, загруженных в процесс, с аналогичным списком другого экземпляра той же программы в другой системе помогает выявить проблемы с конфигурацией DLL, например загрузку в процесс не той версии DLL.

Защита памяти

Как уже говорилось в главе 1, Windows обеспечивает защиту памяти, предотвращая случайную или преднамеренную порчу пользовательскими процессами данных в адресном пространстве системы или других процессов. В Windows предусмотрено четыре основных способа защиты памяти.

Во-первых, доступ ко всем общесистемным структурам данных и пулам памяти, используемым системными компонентами режима ядра, возможен лишь из режима ядра — у потоков пользовательского режима нет доступа к соответствующим страницам. Когда поток пользовательского режима пытается обратиться к одной из таких страниц, процессор генерирует исключение, и диспетчер памяти сообщает потоку о нарушении доступа.

ПРИМЕЧАНИЕ Некоторые страницы в системном адресном пространстве Windows 95/98 и Windows Millennium Edition, напротив, доступны для записи из пользовательского режима, что позволяет сбойным приложениям портить важные системные структуры данных и вызывать крах системы.

Во-вторых, у каждого процесса имеется индивидуальное закрытое адресное пространство, защищенное от доступа потоков других процессов. Исключение составляют те случаи, когда процесс разделяет какие-либо страницы с другими процессами или когда у другого процесса есть права на доступ к объекту «процесс» для чтения и/или записи, что позволяет ему использовать функции *ReadProcessMemory* и *WriteProcessMemory*. Как только поток ссылается на какой-нибудь адрес, аппаратные средства поддержки виртуальной памяти совместно с диспетчером памяти перехватывают это обращение и транслируют виртуальный адрес в физический. Контролируя трансляцию виртуальных адресов, Windows гарантирует, что потоки одного процесса не получат несанкционированного доступа к страницам другого процесса.

В-третьих, кроме косвенной защиты, обеспечиваемой трансляцией виртуальных адресов в физические, все процессоры, поддерживаемые Windows, предоставляют ту или иную форму аппаратной защиты памяти (например доступ для чтения и записи, только для чтения и т. д.); конкретные механизмы такой защиты зависят от архитектуры процессора. Скажем, страницы кода в адресном пространстве процесса помечаются атрибутом «только для чтения», что защищает их от изменения пользовательскими потоками.

Атрибуты защиты памяти, определенные в Windows API, перечислены в таблице 7-3 (см. также документацию на функции *VirtualProtect*, *VirtualProtectEx*, *VirtualQuery* и *VirtualQueryEx*).

Таблица 7-3. *Атрибуты защиты памяти, определенные в Windows API*

Атрибут	Описание
PAGE_NOACCESS	Любая попытка чтения, записи или выполнения кода на этой странице вызывает нарушение доступа
PAGE_READONLY	Любая попытка записи (или выполнения кода на процессорах без поддержки запрета на выполнение) на этой странице вызывает нарушение доступа, но чтение разрешено
PAGE_READWRITE	Страница доступна для чтения и записи — никакие действия не вызывают нарушения доступа
PAGE_EXECUTE	Любая попытка записи на этой странице вызывает нарушение доступа, но выполнение кода (и чтения на всех существующих процессорах) разрешено

Таблица 7-3. (*окончание*)

Атрибут	Описание
PAGE_EXECUTE_READ	Любая попытка записи на этой странице вызывает нарушение доступа, но чтение и выполнение разрешено
PAGE_EXECUTE_READWRITE	Страница доступна для чтения, записи и выполнения — никакие действия не вызывают нарушения доступа
PAGE_WRITECOPY	Любая попытка записи на этой странице заставляет систему выдать процессу закрытую копию данной страницы; при попытке выполнения кода на этой странице возникает нарушение доступа
PAGE_EXECUTE_WRITECOPY	Любая попытка записи на этой странице заставляет систему выдать процессу закрытую копию данной страницы. Чтение и выполнение кода на этой странице разрешено (в этом случае копия страницы не создается)
PAGE_GUARD	Любая попытка чтения или записи сторожевой страницы вызывает исключение EXCEPTION_GUARD_PAGE, и она перестает быть сторожевой; этот атрибут можно комбинировать с любым другим атрибутом, кроме PAGE_NOACCESS
PAGE_NOCACHE	Используется некэшируемая физическая память, что, как правило, не рекомендуется делать. Имеет смысл для драйверов устройств, например, для проецирования буфера видеокадра без кэширования
PAGE_WRITECOMBINE	Разрешает комбинированную запись на страницу памяти. В этом случае процессор может кэшировать запросы на запись в память для большей производительности. Например, при наличии нескольких запросов на запись по одному адресу возможна запись лишь по последнему запросу

ПРИМЕЧАНИЕ Атрибут защиты «запрет на выполнение» (no execute protection) поддерживается Windows XP Service Pack 2 и Windows Server 2003 Service Pack 1 на процессорах с соответствующей аппаратной поддержкой (например, на x64-, IA64- и будущих x86-процессорах). В более ранних версиях Windows и на процессорах без поддержки атрибута защиты «запрет на выполнение», выполнение всегда разрешено. Более подробные сведения об этом атрибуте защиты см. в следующем разделе.

Наконец, совместно используемые объекты «раздел» имеют стандартные для Windows списки контроля доступа (access control lists, ACL), проверяемые при попытках процессов открыть эти объекты. Таким образом, доступ к разделяемой памяти ограничен кругом процессов с соответствующими правами. Когда поток создает раздел для проецирования файла, в этом принимает участие и подсистема защиты. Для создания раздела поток должен иметь права хотя бы на чтение нижележащего объекта «файл», иначе операция закончится неудачно.

После успешного открытия описателя раздела действия потока все равно зависят от описанных выше атрибутов защиты, реализуемых диспетче-

ром памяти на основе аппаратной поддержки. Поток может менять атрибуты защиты виртуальных страниц раздела (на уровне отдельных страниц), если это не противоречит разрешениям, указанным в ACL для данного раздела. Так, диспетчер памяти позволит потоку сменить атрибут страниц общего раздела «только для чтения» на «копирование при записи», но запретит его изменение на атрибут «для чтения и записи». Копирование при записи разрешается потому, что не влияет на другие процессы, тоже использующие эти данные.

Все эти механизмы защиты памяти вносят свой вклад в надежность, стабильность и устойчивость Windows к ошибкам и сбоям приложений.

Запрет на выполнение

Хотя в API управления памятью в Windows всегда были определены биты защиты страницы, позволяющие указывать, может ли страница содержать исполняемый код, лишь с появлением Windows XP Service Pack 2 и Windows Server 2003 Service Pack 1 эта функциональность стала поддерживаться на процессорах с аппаратной защитой «запрет на выполнение», в том числе на всех процессорах AMD64 (AMD Athlon64, AMD Opteron), на некоторых чисто 32-разрядных процессорах AMD (отдельных AMD Sempron), на Intel IA64 и Intel Pentium 4 или Xeon с поддержкой EM64T (Intel Extended Memory 64 Technology).

Эта защита, также называемая предотвращением выполнения данных (data execution prevention, DEP), означает, что попытка передачи управления какой-либо инструкции на странице, помеченной атрибутом «запрет на выполнение», приведет к нарушению доступа к памяти. Благодаря этому блокируются попытки определенных типов вирусов воспользоваться ошибками в операционной системе, которые иначе позволили бы выполнить код, размещенный на странице данных. Попытка выполнить код на странице, помеченной атрибутом «запрет на выполнение», в режиме ядра вызывает крах системы с кодом ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY. Если такая же попытка предпринимается в пользовательском режиме, то генерируется исключение STATUS_ACCESS_VIOLATION (0xc0000005); оно доставляется потоку, в котором была эта недопустимая ссылка. Если процесс выделяет память, которая должна быть исполняемой, то при вызове функций, отвечающих за выделение памяти, он обязан явно указать для соответствующих страниц флаг PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE или PAGE_EXECUTE_WRITECOPY.

В 64-разрядных версиях Windows атрибут защиты «запрет на выполнение» всегда применяется ко всем 64-разрядным программам и драйверам устройств, и его нельзя отключить. Поддержка такой защиты для 32-разрядных программ зависит от конфигурационных параметров системы. В 64-разрядной Windows защита от выполнения применяется к стекам потоков (как режима ядра, так и пользовательского режима), к страницам пользовательского режима, не помеченным явно как исполняемые, к пулу подкачи-

ваемой памяти ядра и к сеансовому пулу ядра (описание пулов памяти ядра см. в разделе «Системные пулы памяти»). Однако в 32-разрядной Windows защита от выполнения применяется только к стекам потоков и страницам пользовательского режима. Кроме того, когда в 32-разрядной Windows разрешена защита от выполнения, система автоматически загружается в PAE-режиме (переходя на использование PAE-ядра, \Windows\System32\Ntkrnlpa.exe). Описание PAE см. в разделе «Physical Address Extension (PAE)».

Активизация защиты от выполнения для 32-разрядных программ зависит от ключа /NOEXECUTE= в Boot.ini. Эти настройки можно изменить и на вкладке Data Execution Prevention, которая открывается последовательным выбором My Computer, Properties, Advanced, Performance Settings (см. рис. 7-2.) Когда вы выбираете защиту от выполнения в диалоговом окне настройки DEP, файл Boot.ini модифицируется добавлением в него соответствующего ключа /NOEXECUTE. Список аргументов для этого ключа и их описание см. в таблице 7-4. 32-разрядные приложения, исключенные из защиты от выполнения, перечисляются в параметрах в разделе реестра HKLM\Software\Microsoft\Windows NT \CurrentVersion\AppCompatFlags\Layers; при этом в качестве имени параметра используется полный путь к исполняемому файлу, а в качестве его значения — «DisableNXShowUI».

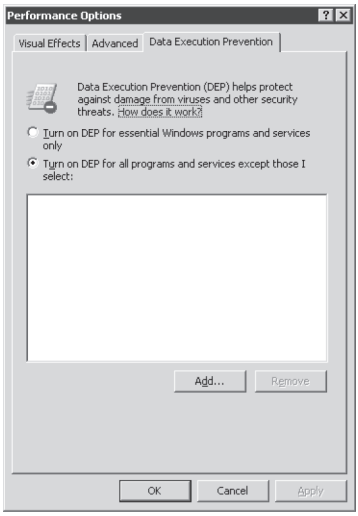


Рис. 7-2. *Параметры Data Execution Protection*

В Windows XP (в 64- и 32-разрядных версиях) защита от выполнения для 32-разрядных программ по умолчанию применяется только к базовым исполняемым образам операционной системы Windows (/NOEXECUTE=OPTIN), чтобы не нарушить работу 32-разрядных приложений, которые могут полагаться на выполнение кода в страницах, не помеченных как исполняемые. В Windows Server 2003 такая защита по умолчанию распространяется на все 32-разрядные приложения (/NOEXECUTE=OPTOUT).

ПРИМЕЧАНИЕ Чтобы получить полный список защищаемых программ, установите Windows Application Compatibility Toolkit (его можно скачать с *www.microsoft.com*) и запустите Compatibility Administrator Tool. Выберите System Database, Applications и Windows Components. В правой секции окна появится список защищенных исполняемых файлов.

Таблица 7-4. *Ключ /NOEXECUTE в Boot.ini*

Ключ	Параметр в диалоговом окне настройки DEP	Описание
/NOEXECUTE=OPTIN	Turn on DEP for necessary Windows programs and services only (включение DEP только для необходимых программ и сервисов Windows)	Включает DEP для базовых системных образов Windows
/NOEXECUTE=OPTOUT	Turn on DEP for all programs and services except those that I select (включение DEP для всех программ и сервисов, кроме выбранных мной)	Включает DEP для всех исполняемых образов, кроме указанных
/NOEXECUTE=ALWAYSON	(Этот вариант в GUI не представлен)	Включает DEP для всех компонентов без возможности исключения определенных приложений
/NOEXECUTE=ALWAYSOFF	(Этот вариант в GUI не представлен)	Отключает DEP (не рекомендуется)

Программный вариант DEP

Поскольку большинство процессоров, на которых сегодня работает Windows, не поддерживает аппаратную защиту от выполнения, Windows XP Service Pack 2 и Windows Server 2003 Service Pack 1 (или выше) поддерживают ограниченный программный вариант DEP (data execution prevention). Одна из функций программного DEP — сужать возможности злоумышленников в использовании механизма обработки исключений в Windows. (Описание структурной обработки исключений см. в главе 3.) Если файлы образа программы опираются на безопасную структурную обработку исключений (новая функциональность компилятора Microsoft Visual C++ 2003), то, прежде чем передавать исключение, система проверяет, зарегистрирован ли обработчик этого исключения в таблице функций, которая помещается в файл образа. Если в файлах образа программы безопасная структурная обработка исключений не применяется, программный DEP проверяет, находится ли обработчик исключения в области памяти, помеченной как исполняемая, еще до передачи исключения.

Копирование при записи

Защита страницы типа «копирование при записи» — механизм оптимизации, используемый диспетчером памяти для экономии физической памяти. Когда процесс проецирует копируемое при записи представление объекта «раздел» со страницами, доступными для чтения и записи, диспетчер памяти — вместо того чтобы создавать закрытую копию этих страниц в момент проецирования представления (как в операционной системе Hewlett Packard OpenVMS) — откладывает создание копии до тех пор, пока не закончится запись в них. Эта методика используется и всеми современными UNIX-системами. На рис. 7-3 показана ситуация, когда два процесса совместно используют три страницы, каждая из которых помечена как копируемая при записи, но ни один из процессов еще не пытался их модифицировать.

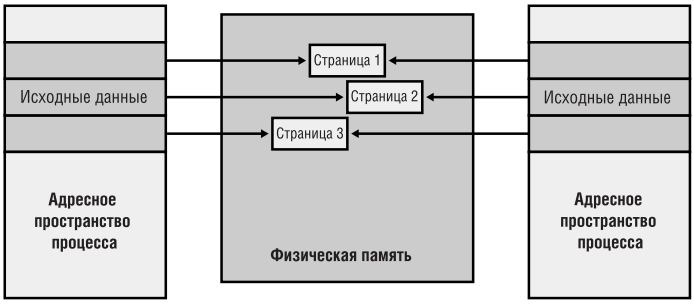


Рис. 7-3. *Состояние страниц до копирования при записи*

Если поток любого из этих процессов что-то записывает на такую страницу, генерируется исключение, связанное с управлением памятью. Обнаружив, что запись ведется на страницу с атрибутом «копирование при записи», диспетчер памяти, вместо того чтобы сообщить о нарушении доступа, выделяет в физической памяти новую страницу, доступную для чтения и записи, копирует в нее содержимое исходной страницы, обновляет соответствующую информацию о страницах, проецируемых на данный процесс, и закрывает исключение. В результате команда, вызвавшая исключение, выполняется повторно, и операция записи проходит успешно. Но, как показано на рис. 7-4, новая страница теперь является личной собственностью процесса, инициировавшего запись, и не видима другим процессам, совместно использующим страницу с атрибутом «копирование при записи». Каждый процесс, что-либо записывающий на эту разделяемую страницу, получает в свое распоряжение ее закрытую копию.

Одно из применений копирования при записи — поддержка точек прерываний для отладчиков. Например, по умолчанию страницы кода доступны только для выполнения. Если программист при отладке программы устанавливает точку прерывания, отладчик должен добавить в код программы соответствующую команду. Для этого он сначала меняет атрибут защиты страницы на PAGE_EXECUTE_READWRITE, а затем модифицирует поток команд. Поскольку страница кода является частью проецируемого раздела,

диспетчер памяти создает закрытую копию для процесса с установленной точкой прерывания, тогда как другие процессы по-прежнему используют исходную страницу кода.

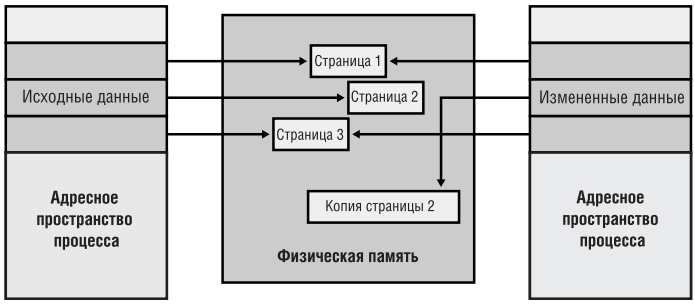


Рис. 7-4. *Состояние страниц после копирования при записи*

Копирование при записи может служить примером алгоритма *отложенной оценки* (lazy evaluation), который диспетчер памяти применяет при любой возможности. В таких алгоритмах операции, чреватые большими издержками, не выполняются до тех пор, пока не станут абсолютно необходимыми, — если операция так и не понадобится, никаких издержек вообще не будет.

Подсистема POSIX использует преимущества копирования при записи в реализации функции *fork*. Как правило, если UNIX-приложения вызывают *fork* для создания другого процесса, то первое, что делает новый процесс, — обращается к функции *exec* для повторной инициализации адресного пространства исполняемой программы. Вместо копирования всего адресного пространства при вызове *fork* новый процесс использует страницы родительского процесса, помечая их как копируемые при записи. Если дочерний процесс что-то записывает на эти страницы, он получает их закрытую копию. В ином случае оба процесса продолжают разделять страницы без копирования. Так или иначе диспетчер памяти копирует лишь те страницы, на которые процесс пытается что-то записать, а не все содержимое адресного пространства.

Оценить частоту срабатывания механизма копирования при записи можно с помощью счетчика Memory: Write Copies/Sec (Память: Запись копий страниц/сек).

Диспетчер куч

Многие приложения выделяют память небольшими блоками (менее 64 Кб — минимума, поддерживаемого функциями типа *VirtualAlloc*). Выделение столь большой области (64 Кб) для сравнительно малого блока весьма неоптимально с точки зрения использования памяти и производительности. Для устранения этой проблемы в Windows имеется компонент — *диспетчер куч* (heap manager), который управляет распределением памяти внутри больших областей, зарезервированных с помощью функций, выделяющих память в соответствии с гранулярностью страниц. Гранулярность выделения памяти в диспетчере куч сравнительно мала: 8 байтов в 32-разрядных системах и 16 байтов в

64-разрядных. Диспетчер куч обеспечивает оптимальное использование памяти и производительность при выделении таких небольших блоков памяти.

Функции диспетчера куч локализованы в двух местах: в Ntdll.dll и Ntoskrnl.exe. API-функции подсистем (вроде API-функций Windows-куч) вызывают функции из Ntdll, а компоненты исполнительной системы и драйверы устройств — из Ntoskrnl. Родные интерфейсы (функции с префиксом *Rtl*) доступны только внутренним компонентам Windows и драйверам устройств режима ядра. Документированный интерфейс Windows API для куч (функции с префиксом *Heap*) представляют собой тонкие оболочки, которые вызывают родные функции из Ntdll.dll. Кроме того, для поддержки устаревших Windows-приложений предназначены унаследованные API-функции (с префиксом *Local* или *Global*). К наиболее часто используемым Windows-функциям куч относятся:

- *HeapCreate* или *HeapDestroy* — соответственно создает или удаляет кучу. При создании кучи можно указать начальные размеры зарезервированной и переданной памяти;
- *HeapAlloc* — выделяет блок памяти из кучи;
- *HeapFree* — освобождает блок, ранее выделенный через *HeapAlloc*;
- *HeapReAlloc* — увеличивает или уменьшает размер уже выделенного блока;
- *HeapLock* и *HeapUnlock* — управляют взаимным исключением (mutual exclusion) операций, связанных с обращением к куче;
- *HeapWalk* — перечисляет записи и области в куче.

Типы куч

У каждого процесса имеется минимум одна куча — куча, выделяемая процессу по умолчанию (default process heap). Куча по умолчанию создается в момент запуска процесса и никогда не удаляется в течение срока жизни этого процесса. По умолчанию она имеет размер 1 Мб, но ее начальный размер может быть увеличен, если в файле образа указано иное значение с помощью ключа /HEAP компоновщика. Однако этот объем памяти резервируется только для начала и по мере необходимости автоматически увеличивается (в файле образа можно указать и начальный размер переданной памяти).

Куча по умолчанию может быть явно использована программой или неявно некоторыми внутренними Windows-функциями. Приложение запрашивает память из кучи процесса по умолчанию вызовом Windows-функции *GetProcessHeap*. Процесс может создавать дополнительные закрытые кучи вызовом *HeapCreate*. Когда куча больше не нужна, занимаемое ею виртуальное адресное пространство можно освободить, вызвав *HeapDestroy*. Массив всех куч поддерживается в каждом процессе, и поток может обращаться к ним через Windows-функцию *GetProcessHeaps*.

Куча может быть создана в больших регионах памяти, зарезервированных через диспетчер памяти с помощью *VirtualAlloc* или через объекты «файл, проецируемый в память», отображенные на адресное пространство

процесса. Последний подход редко применяется на практике, но удобен в случаях, когда содержимое блоков нужно разделять между двумя процессами или между частями компонента, работающими в режиме ядра и в пользовательском режиме. В последнем случае действует ряд ограничений на вызов функций куч. Во-первых, внутренние структуры куч используют указатели и поэтому не допускают перемещения на другие адреса. Во-вторых, функции куч не поддерживают синхронизацию между несколькими процессами или между компонентом ядра и процессом пользовательского режима. Наконец, в случае кучи, разделяемой между кодом пользовательского режима и режима ядра проекция пользовательского режима должна быть только для чтения, чтобы исключить повреждение внутренних структур кучи кодом пользовательского режима.

Структура диспетчера кучи

Как показано на рис. 7-5, диспетчер куч состоит из двух уровней: необязательного интерфейсного (front-end layer) и базового (core heap layer). Последний заключает в себе базовую функциональность, которая обеспечивает управление блоками внутри сегментов, управление сегментами, поддержку политик расширения кучи, передачу и возврат памяти, а также управление большими блоками.

Необязательный интерфейсный уровень (только для куч пользовательского режима) размещается поверх базового уровня. Существует два типа интерфейсных уровней: ассоциативные списки (look-aside lists) и куча с малой фрагментацией (Low Fragmentation Heap, LFH). LFH доступна лишь в Windows XP и более поздних версиях Windows. Единовременно для каждой кучи можно использовать только один интерфейсный уровень.

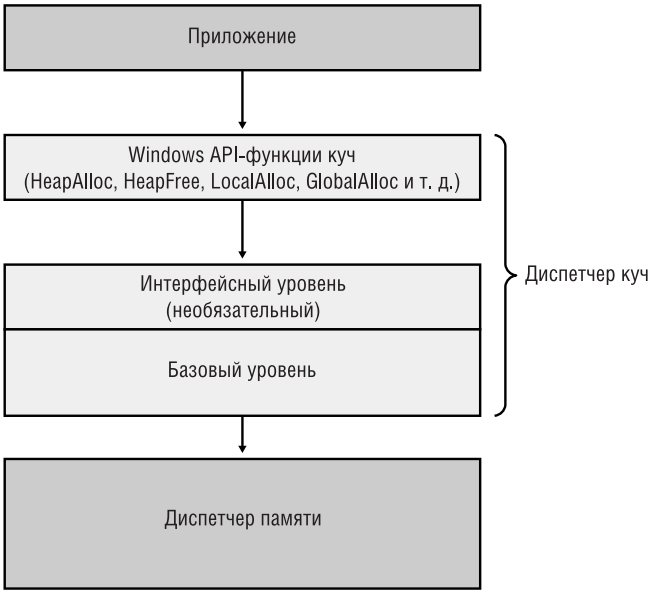


Рис. 7-5. *Уровни диспетчера куч*

Синхронизация доступа к куче

Диспетчер куч по умолчанию поддерживает параллельный доступ из нескольких потоков. Однако, если процесс является однопоточным или использует внешний механизм синхронизации, он может отключить синхронизацию, поддерживаемую диспетчером куч, и тем самым избежать издержек, связанных с этим видом синхронизации. Для этого при создании кучи или при каждом запросе на выделение памяти такой процесс может указывать флаг HEAP_NO_SERIALIZE.

Процесс также может блокировать всю кучу и запретить другим потокам выполнение операций, требующих согласования состояний между несколькими обращениями к куче. Например, перечисление блоков в куче с помощью Windows-функции *HeapWalk* требует блокировки кучи, если над ней могут выполняться операции сразу несколькими потоками.

Если синхронизация куч разрешена, для каждой кучи выделяется по одной блокировке, которая защищает все внутренние структуры кучи. В приложениях с большим числом потоков (особенно когда они выполняются в многопроцессорных системах) блокировка кучи может превратиться в точку интенсивной конкуренции. В таком случае производительность можно повысить за счет использования интерфейсного уровня.

Ассоциативные списки

Это однонаправленные связанные списки (single linked lists), поддерживающие элементарные операции вроде заталкивания в список или выталкивания из него по принципу «последним пришел, первым вышел» (Last In, First Out, LIFO) без применения блокирующих алгоритмов. Упрощенная версия этих структур данных также доступна Windows-приложениям через функции *InterlockedPopEntrySList* и *InterlockedPushEntrySList*. Для каждой кучи создается 128 ассоциативных списков, которые удовлетворяют запросы на выделение блоков памяти размером до 1 Кб на 32-разрядных платформах и до 2 Кб на 64-разрядных.

Ассоциативные списки обеспечивают гораздо более высокую производительность, чем при обычных запросах на выделение памяти, так как несколько потоков могут одновременно выполнять операции выделения и возврата памяти, не требуя применения глобальной для кучи блокировки. Кроме того, благодаря модели размещения LIFO и обращению к меньшему числу внутренних структур данных при каждой операции над кучей оптимизируется локальность кэша.

Диспетчер куч поддерживает ряд блокировок в каждом ассоциативном списке и некоторые счетчики, помогающие независимо регулировать работу с каждым списком. Если поток запрашивает блок такого размера, которого нет в соответствующем ассоциативном списке, диспетчер куч переадресует этот вызов базовому уровню и обновит внутренний счетчик неудачных выделений, значение которого впоследствии будет использовано при принятии решений по оптимизации.

Диспетчер куч создает ассоциативные списки автоматически при создании кучи, если только эта куча расширяемая и не включен отладочный режим. У некоторых приложений могут возникать проблемы совместимости из-за использования диспетчером куч ассоциативных списков. В таких случаях для корректной работы нужно указывать флаг *DisableHeapLookaside* в параметрах выполнения файлов образов унаследованных приложений. (Эти параметры можно задавать с помощью утилиты Imagecfg.exe из Windows 2000 Server Resource Kit, supplement 1.)

Куча с малой фрагментацией

Многие приложения, выполняемые в Windows, используют сравнительно небольшие объемы памяти из куч (обычно менее одного мегабайта). Для этого класса приложений диспетчер куч применяет политику наибольшей подгонки (best-fit policy), которая помогает сохранять небольшим «отпечаток» каждого процесса в памяти. Однако такая стратегия не масштабируется для больших процессов и многопроцессорных машин. В этих случаях доступная память в куче может уменьшиться из-за ее фрагментации. В сценариях, где лишь блоки определенного размера часто используются параллельно разными потоками, выполняемыми на разных процессорах, производительность ухудшается. Дело в том, что нескольким процессорам нужно одновременно модифицировать один и тот же участок памяти (например, начало ассоциативного списка для блоков этого размера), а это приводит к объявлению недействительной соответствующей кэш-линии для других процессоров.

Эти проблемы решаются применением кучи с малой фрагментацией (LFH), которая использует базовый уровень диспетчера куч и ассоциативные списки. В отличие от ситуации, в которой ассоциативные списки по умолчанию применяются как интерфейсные, если это разрешено другими параметрами куч, поддержка LFH включается, только когда приложение вызывает функцию *HeapSetInformation*. В случае больших куч значительная доля запросов на выделение обычно раскладывается на относительно небольшое число корзин (buckets) определенных размеров. Стратегия выделения памяти, применяемая LFH, заключается в оптимизации использования памяти для таких запросов за счет эффективной обработки блоков одного размера.

Для устранения проблем с масштабируемостью LFH раскрывает часто используемые внутренние структуры в набор слотов, в два раза больший текущего количества процессоров в компьютере. Закрепление потоков за этими слотами выполняется LFH-компонентом, называемым *диспетчером привязки* (affinity manager). Изначально LFH использует для распределения памяти первый слот, но, как только возникает конкуренция при доступе к некоторым внутренним данным, переключает текущий поток на другой слот. И чем больше конкуренция, тем большее число слотов задействуется для потоков. Эти слоты создаются для корзины каждого размера, что также увеличивает локальность и сводит к минимуму общий расход памяти.

Средства отладки

Диспетчер куч предоставляет несколько средств, помогающих обнаруживать ошибки.

- **Enable tail checking (включить проверку концевой части блока)** В конец каждого блока помещается сигнатура, проверяемая при его освобождении. Если эта сигнатура полностью или частично уничтожается из-за переполнения буфера, куча сообщает о соответствующей ошибке.
- **Enable free checking (включить проверку свободных блоков)** Свободный блок заполняется определенным шаблоном, который проверяется, когда диспетчеру куч нужен доступ к этому блоку. Если процесс продолжает записывать в блок после его освобождения, диспетчер куч обнаружит изменения в шаблоне и сообщит об ошибке.
- **Parameter checking (проверка параметров)** Проверка параметров, передаваемых функциям куч.
- **Heap validation (проверка кучи)** Вся куча проверяется при каждом обращении к ней.
- **Heap tagging and stack traces support (поддержка меток и трассировки стека)** Это средство поддерживает задание меток для выделяемой памяти и/или перехват трассировок стека пользовательского режима при обращениях к куче, что помогает локализовать причину той или иной ошибки.

Первые три средства включаются по умолчанию, если загрузчик обнаруживает, что процесс запущен под управлением отладчика. (Отладчик может переопределить такое поведение и выключить эти средства.) Средства отладки для куч могут быть заданы установкой различных отладочных флагов в заголовке образа через утилиту gflags (см. раздел «Глобальные флаги Windows» в главе 3) или командой *!heap* в любом стандартном отладчике Windows.

Включение средств отладки влияет на все кучи в процессе. Кроме того, включение любого средства отладки приводит к автоматическому отключению интерфейсного уровня и переходу на использование базового. Интерфейсный уровень также не применяется для нерасширяемых куч (из-за дополнительных издержек) или для куч, не допускающих сериализации.

Pageheap

Так как при проверке концевых частей блоков и шаблона свободных блоков могут обнаруживаться повреждения, произошедшие задолго до проявления собственно проблемы, предоставляется дополнительный инструмент отладки куч, *pageheap*, который переадресует все обращения к куче (или их часть) другому диспетчеру куч. Pageheap является частью Windows Application Compatibility Toolkit, и его можно скачать с *www.microsoft.com*. Pageheap помещает выделенные блоки в конец страниц, поэтому при переполнении буфера возникнет нарушение доступа, что упростит выявление ошибочного кода. Блоки можно помещать и в начало страниц для обнаружения проблем, свя-

занных с неполным использованием буферов (buffer underruns). (Такие ситуации — большая редкость.) Pageheap также позволяет защищать освобожденные страницы от любых видов доступа для выявления ссылок на блоки после их освобождения.

Заметьте, что применение pageheap может привести к нехватке адресного пространства, так как выделение даже очень малых блоков памяти сопряжено с существенными издержками. Также может ухудшиться производительность из-за увеличения количества ссылок на обнуленные страницы, потери локальности и частых вызовов для проверки структур кучи. Чтобы уменьшить негативное влияние на производительность, pageheap можно использовать только для блоков определенных размеров, конкретных диапазонов адресов и т. д.

ПРИМЕЧАНИЕ Подробнее о pageheap см. статью 286470 в Microsoft Knowledge Base (*http://support.microsoft.com*).

Address Windowing Extensions

Хотя 32-разрядные версии Windows поддерживают до 128 Гб физической памяти (см. таблицу 2-4 в главе 2), размер виртуального адресного пространства любого 32-разрядного пользовательского процесса по умолчанию равен 2 Гб (при указании загрузочных параметров /3GB и /USERVA в Boot.ini этот размер составляет 3 Гб). Чтобы 32-разрядный процесс мог получить доступ к большему объему физической памяти, Windows поддерживает набор функций под общим названием Address Windowing Extensions (AWE). Так, в системе под управлением Windows 2000 Advanced Server с 8 Гб физической памяти серверное приложение базы данных может с помощью AWE использовать под кэш базы данных до 6 Гб памяти.

Выделение и использование памяти через функции AWE осуществляется в три этапа.

1. Выделение физической памяти.
2. Создание региона виртуального адресного пространства — окна, на которое будут проецироваться представления физической памяти.
3. Проецирование на окно представлений физической памяти.

Для выделения физической памяти приложение вызывает Windows-функцию *AllocateUserPhysicalPages*. (Эта функция требует, чтобы у пользователя была привилегия Lock Pages In Memory.) Затем приложение обращается к Windows-функции *VirtualAlloc* с флагом MEM_PHYSICAL, чтобы создать окно в закрытой части адресного пространства процесса, на которое проецируется (частично или полностью) ранее выделенная физическая память. Память, выделенная через AWE, может быть использована почти всеми функциями Windows API. (Например, функции Microsoft DirectX ее не поддерживают.)

Если приложение создает в своем адресном пространстве окно размером 256 Мб и выделяет 4 Гб физической памяти (в системе с объемом физической памяти более 4 Гб), то оно получает доступ к любой части физической

памяти, проецируя ее на это окно через Windows-функции *MapUserPhysical-Pages* или *MapUserPhysicalPagesScatter*. Размер физической памяти, единовременно доступный приложению при такой схеме выделения, определяется размером окна в виртуальном адресном пространстве. На рис. 7-6 показано AWE-окно в адресном пространстве серверного приложения, на которое проецируется регион физической памяти, предварительно выделенный через *AllocateUserPhysicalPages*.

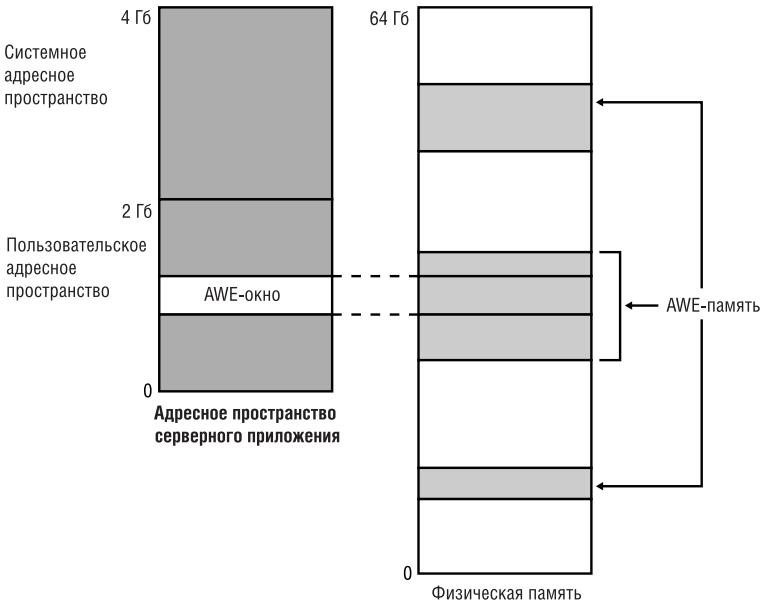


Рис. 7-6. *Использование AWE для проецирования физической памяти*

AWE-функции имеются во всех выпусках Windows и доступны независимо от объема физической памяти в системе. Однако AWE наиболее полезен в системах с объемом физической памяти не менее 2 Гб, поскольку тогда этот механизм — единственное средство для прямого использования более чем 2 Гб памяти 32-разрядным процессом. Еще одно его применение — защита. Так как AWE-память никогда не выгружается на диск, данные в этой памяти никогда не имеют копии в страничном файле, а значит, никто не сумеет просмотреть их, загрузив компьютер с помощью альтернативной операционной системы.

Теперь несколько слов об ограничениях, налагаемых на память, которая выделяется и проецируется с помощью AWE-функций.

- Страницы такой памяти нельзя разделять между процессами.
- Одну и ту же физическую страницу нельзя проецировать по более чем одному виртуальному адресу в рамках одного процесса.
- В более старых версиях Windows страницы такой памяти могут иметь единственный атрибут защиты — «для чтения и записи». В Windows Server

2003 Service Pack 1 и выше также поддерживаются атрибуты «нет доступа» и «только для чтения».

О структурах данных таблицы страниц, используемой для проецирования памяти в системах с более чем 4 Гб физической памяти, см. раздел «Physical Address Extension (PAE)» далее в этой главе.

Системные пулы памяти

При инициализации системы диспетчер памяти создает два типа динамических пулов памяти, используемых компонентами режима ядра для выделения системной памяти.

- **Пул неподкачиваемой памяти (nonpaged pool)** Состоит из диапазонов системных виртуальных адресов, которые всегда присутствуют в физической памяти и доступны в любой момент (при любом IRQL и из контекста любого процесса) без генерации ошибок страниц. Одна из причин существования такого пула — невозможность обработки ошибок страниц при IRQL уровня «DPC/dispatch» и выше (см. главу 2).
- **Пул подкачиваемой памяти (paged pool)** Регион виртуальной памяти в системном пространстве, содержимое которого система может выгружать в страничный файл и загружать из него. Драйверы, не требующие доступа к памяти при IRQL уровня «DPC/dispatch» и выше, могут использовать память из этого пула. Он доступен из контекста любого процесса.

Оба пула находятся в системном адресном пространстве и проецируются на виртуальное адресное пространство любого процесса (их начальные адреса в системной памяти перечислены в таблице 7-8). Исполнительная система предоставляет функции для выделения и освобождения памяти в этих пулах (см. описание функций, чьи имена начинаются с *ExAllocatePool*, в Windows DDK).

В однопроцессорных системах создается три пула подкачиваемой памяти, а в многопроцессорных — пять. Наличие нескольких подкачиваемых пулов уменьшает частоту блокировки системного кода при одновременных обращениях нескольких потоков к процедурам управления пулами. Начальный размер подкачиваемого и неподкачиваемого пулов зависит от объема физической памяти и может при необходимости расти до максимального значения, вычисляемого в период загрузки системы.

ПРИМЕЧАНИЕ Будущие выпуски Windows, возможно, будут поддерживать пулы динамических размеров, а значит, лимита на максимальный размер больше не будет. Таким образом, в приложениях и драйверах устройств нельзя исходить из того, что максимальный размер пула является фиксированной величиной в любой системе.

Настройка размеров пулов

Чтобы установить другие начальные размеры этих пулов, измените значения параметров NonPagedPoolSize и PagedPoolSize в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management с 0

(при этом система сама вычисляет размеры) на нужные величины (в байтах). Но вы не сможете превысить предельные значения, перечисленные в таблице 7-5. Значение 0xFFFFFFFF для PagedPoolSize указывает, что выбран наибольший из возможных размеров, однако увеличение пула подкачиваемой памяти будет происходить за счет записей системной таблицы страниц (page table entries, PTE).

Таблица 7-5. *Максимальные размеры пулов*

Тип пула	Максимальный размер в 32-разрядных системах	Максимальный размер в 64-разрядных системах
Неподкачиваемый	256 Мб (128 Мб, если был задан загрузочный параметр /3GB)	128 Гб
Подкачиваемый	491,875 Мб (в Windows 2000 и Windows XP); 650 Мб (Windows Server 2003)	128 Гб

Рассчитанные значения размеров хранятся в четырех переменных ядра, три из которых экспортируются как счетчики производительности. Имена переменных, счетчиков и параметров реестра, позволяющих изменять размеры пулов, перечислены в таблице 7-6.

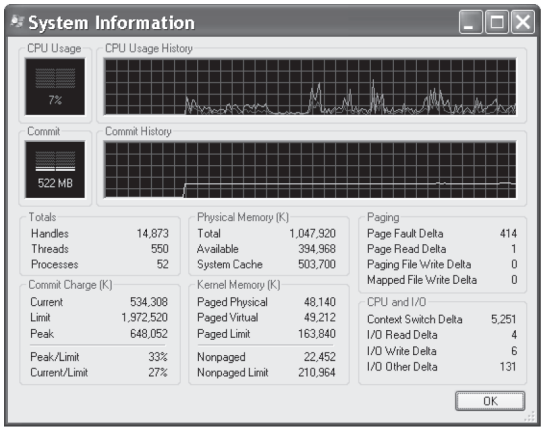
Таблица 7-6. *Переменные и счетчики производительности, отражающие размеры системных пулов*

Переменная ядра	Счетчик производительности	Раздел реестра	Описание
MmSizeOfNonPagedPoolInBytes	Memory: Pool Nonpaged Bytes (Память: Байт в невыгружаемом страничном пуле)	Нет	Текущий размер пула неподкачиваемой памяти
MmMaximumNonPagedPoolInBytes	Нет	HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\NonPagedPoolSize	Максимальный размер пула неподкачиваемой памяти
Нет	Memory: Pool Paged Bytes (Память: Байт в выгружаемом страничном пуле)	Нет	Текущий размер виртуальной части пула подкачиваемой памяти
MmPagedPoolPage (число страниц)	Memory: Pool Paged Resident Bytes (Память: байт в резидентном страничном пуле)	Нет	Текущий размер физической (резидентной) части пула подкачиваемой памяти
MmSizeOfPagedPool-InBytes	Нет	HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagedPoolSize	Максимальный размер виртуальной части пула подкачиваемой памяти

ЭКСПЕРИМЕНТ: определяем максимальные размеры пулов

Поскольку пулы подкачиваемой и неподкачиваемой памяти являются критическими ресурсами системы, важно знать, когда их размер приближается к расчетному для вашей системы пределу, чтобы задать значение, отличное от установленного по умолчанию в соответствующих параметрах реестра. Счетчики выводят лишь текущий, но не максимальный размер, поэтому вы не узнаете о приближении к лимиту, пока не достигнете его. (Как уже говорилось, будущие версии Windows, возможно, будут поддерживать пулы динамических размеров. И тогда необходимость в проверке максимальных размеров пулов отпадет.)

Получить максимальные размеры пулов можно с помощью Process Explorer или отладки ядра в работающей системе (см. главу 1). Для просмотра этих данных через Process Explorer, щелкните View, System Information. Максимальные размеры пулов показываются в секции Kernel Memory, как на следующей иллюстрации.



Заметьте: чтобы Process Explorer мог получить эту информацию, у него должен быть доступ к символам для ядра данной системы. (Как настроить Process Explorer на использование символов, см. в эксперименте «Просмотр детальных сведений о процессах с помощью Process Explorer» в главе 1.)

Для просмотра той же информации в отладчике ядра используйте команду *!vm*:

```
lkd> !vm

*** Virtual Memory Usage ***
Physical Memory:    261980   ( 1047920 Kb)
Page File: \??\C:\pagefile.sys
   Current:   1024000Kb Free Space:    862236Kb
   Minimum:   1024000Kb Maximum:      1024000Kb
Available Pages:     96077   (  384308 Kb)
```

```
ResAvail Pages:     194722   (  778888 Kb)
Locked IO Pages:        89   (     356 Kb)
Free System PTEs:   175933   (  703732 Kb)
Free NP PTEs:        31832   (  127328 Kb)
Free Special NP:         0   (       0 Kb)
Modified Pages:        529   (    2116 Kb)
Modified PF Pages:     529   (    2116 Kb)
NonPagedPool Usage:   5617   (   22468 Kb)
NonPagedPool Max:    52741   (  210964 Kb)
PagedPool 0 Usage:    6582   (   26328 Kb)
PagedPool 1 Usage:    3009   (   12036 Kb)
PagedPool 2 Usage:    3044   (   12176 Kb)
PagedPool Usage:     12635   (   50540 Kb)
PagedPool Maximum:   40960   (  163840 Kb)
```

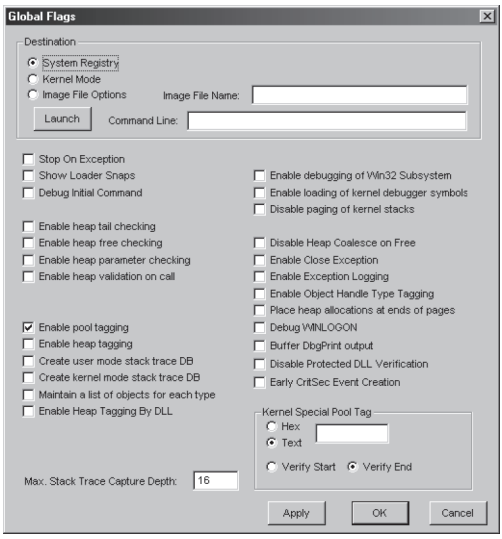
В этой системе размеры пулов подкачиваемой и неподкачиваемой памяти далеки от своих максимумов. Отладчик ядра также позволяет изучить значения переменных ядра, перечисленных в таблице 7-6:

```
kd> dd mmmaximumnonpagedpoolinbytes l1
8047f620  0328c000
kd> ? 328c000
Evaluate expression: 53002240 = 0328c000
kd> dd mmsizeofpagedpoolinbytes l1
80470a98  06800000
kd> ? 6800000
Evaluate expression: 109051904 = 06800000
```

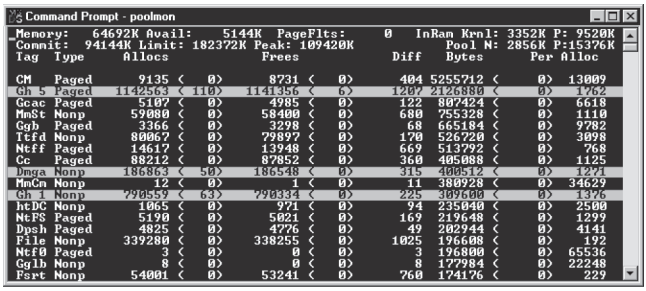
Из этого примера видно, что максимальный размер неподкачиваемого пула составляет 53 002 240 байтов (примерно 50 Мб), а максимальный размер подкачиваемого пула — 109 051 904 байта (104 Мб). В тестовой системе, использованной нами для этого эксперимента, текущий размер использованной памяти неподкачиваемого пула составлял 5,5 Мб, а подкачиваемого пула — 34 Мб, так что оба пула были далеки от заполнения.

Мониторинг использования пулов

Объект Memory (Память) предоставляет отдельные счетчики размеров пулов неподкачиваемой и подкачиваемой памяти (как для виртуальной, так и для физической частей). Кроме того, утилита Poolmon из Windows Support Tools сообщает детальную информацию об использовании этих пулов. Для просмотра такой информации нужно включить внутренний параметр Enable Pool Tagging (который всегда включен в проверочных версиях, а также в Windows Server 2003, где его вообще нельзя выключить). Чтобы включить данный параметр, запустите утилиту Gflags из Windows Support Tools, Platform SDK или DDK и выберите переключатель Enable Pool Tagging, как показано ниже.



Теперь щелкните кнопку Apply и перезагрузите систему. После перезагрузки запустите Poolmon. При этом вы должны увидеть примерно следующее.



Строки с меняющимися данными выделяются подсветкой. (Ее можно выключить, введя букву *l* в окне Poolmon. Повторный ввод *l* вновь включает подсветку.) Нажав клавишу со знаком вопроса в Poolmon, можно просмотреть справочный экран. Вы можете указать пулы, за которыми хотите наблюдать (только подкачиваемый, только неподкачиваемый или и то, и другое), а также порядок сортировки. Кроме того, на справочном экране поясняются параметры командной строки, позволяющие наблюдать за конкретными структурами (или за всеми структурами, но одного типа). Так, команда *poolmon -iCM* позволит следить только за структурами типа CM (которые принадлежат диспетчеру конфигурации, управляющему реестром). Колонки, в которых программа выводит свою информацию, описаны в таблице 7-7.

Таблица 7-7. *Колонки в Poolmon*

Колонка	Описание
Tag	Четырехбайтовая метка, которая присвоена данной области, выделенной из пула
Type	Тип пула (подкачиваемый или неподкачиваемый)
Allocs	Счетчик всех выделений из пула (число в скобках сообщает разницу между текущим значением в колонке Allocs и значением на момент последнего обновления)
Frees	Счетчик всех освобождений (число в скобках сообщает разницу между текущим значением в колонке Frees и значением на момент последнего обновления)
Diff	Разница между Allocs и Frees
Bytes	Суммарное число байтов, занятых структурами этого типа (число в скобках сообщает разницу между текущим значением в колонке Bytes и значением на момент последнего обновления)
Per Alloc	Размер одного экземпляра данной структуры (в байтах)

В этом примере структуры CM занимают основную часть пула подкачиваемой памяти, а структуры MmSt (структуры, относящиеся к управлению памятью и используемые для проецируемых файлов) — основную часть пула неподкачиваемой памяти.

Описание меток пулов см. в файле \Program Files\Debugging Tools for Windows\Triage\Pooltag.txt. (Он устанавливается вместе с Windows Debugging Tools.) Поскольку в этом файле не перечислены метки пулов для сторонних драйверов устройств, используйте ключ *–c* в версии Poolmon, поставляемой с Windows Server 2003 Device Driver Kit (DDK), для генерации файла меток локальных пулов (Localtag.txt). В этом файле содержатся метки пулов, используемых любыми драйверами, которые были обнаружены в вашей системе. (Учтите: если двоичный файл драйвера устройства был удален после загрузки, метки его пулов не распознаются.)

В качестве альтернативы можно вести поиск драйверов устройств в системе по метке пула, используя утилиту Strings.exe с сайта *www.sysinternals.com*. Например, команда:

```
strings \windows\system32\drivers\*.sys > findstr /i "abcd"
```

покажет драйверы, содержащие строку «abcd». Заметьте, что драйверы устройств не обязательно должны находиться в \Windows\System32\Drivers — они могут быть в любом каталоге. Чтобы перечислить полные пути всех загруженных драйверов, откройте меню Start (Пуск), выберите команду Run (Выполнить) и введите **Msinfo32**. Потом щелкните Software Environment (Программная среда) и System Drivers (Системные драйверы).

Еще один способ для просмотра использования пулов драйвером устройства — включение наблюдения за пулами в Driver Verifier (см. далее в этой главе). Хотя при этом способе сопоставление метки пула с драйвером не нужно, он требует перезагрузки (чтобы включить функциональность наблю-

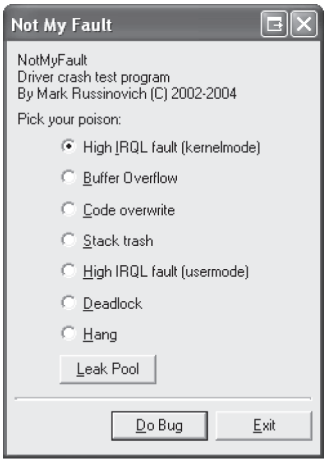
дения за пулами в Driver Verifier для интересующих вас драйверов). После этого вы можете либо запустить Driver Verifier Manager (\Windows\System32\Verifier.exe), либо использовать команду Verifier /Log для записи информации об использовании пулов в какой-либо файл.

Наконец, если вы изучаете аварийный дамп, то можете исследовать использование пулов и с помощью команды *!poolused*. Команда *!poolused 2* сообщает об использовании пула неподкачиваемой памяти с сортировкой по структурам, занимающим наибольшее количество памяти, а команда *!poolused 4* — об использовании пула подкачиваемой памяти (с той же сортировкой). Ниже приведен фрагмент выходной информации этих двух команд.

```
lkd> !poolused 2
  Sorting by  NonPaged Pool Consumed
 Pool Used:
           NonPaged            Paged
 Tag    Allocs     Used    Allocs     Used
 MmCm      386  2237440         0        0    Calls made to
MmAllocateContiguousMemory, Binary: nt!mm
 File    12544  2121232         0        0    File objects
 Devi      385  1487184         0        0    Device objects
 Ntfr    19163  1226888         0        0    ERESOURCE,
Binary: ntfs.sys

kd> !poolused 4
  Sorting by Paged Pool Consumed
 Pool Used:
           NonPaged            Paged
 Tag    Allocs     Used    Allocs     Used
 CM         11      704      5146 22311904
 Gh 5        0        0       727  2523648
 MmSt        0        0       968  1975872
 Ntff       10     2240       790   682560
 Gla1        1      128       383   600544
 Ttfd        0        0       265   545440
```

ЭКСПЕРИМЕНТ: анализ утечки памяти в пуле

В этом эксперименте вы устраните реальную утечку в пуле подкачиваемой памяти в своей системе, чтобы научиться на практике применять способы, описанные в предыдущем разделе. Утечка будет создаваться утилитой NotMyFault, которую можно скачать по ссылке *www.sysinternals.com/windowsinternals.shtml*. (Заметьте, что эта утилита отсутствует в списке инструментов на основной странице Sysinternals.) После запуска NotMyFault.exe загружает драйвер устройства Myfault.sys и выводит такое диалоговое окно.



1. Щелкните кнопку Leak Pool. Это заставит NotMyFault посылать запросы драйверу устройства Myfault на выделение памяти из подкачиваемого пула. (Не нажимайте кнопку Do Bug, иначе вы вызовете крах системы; предназначение этой кнопки описывается в главе 14, где демонстрируются различные типы аварийных ситуаций.) NotMyFault продолжит посылать запросы, пока вы не щелкнете кнопку Stop Leaking. Заметьте, что пул подкачиваемой памяти не освобождается даже при закрытии программы; в нем происходит постоянная утечка памяти до перезагрузки системы. Однако, поскольку утечка пула будет непродолжительной, это не должно вызвать никаких проблем в вашей системе.
2. Пока происходит утечка памяти в пуле, сначала откройте диспетчер задач и перейдите на вкладку Performance (Быстродействие). Вы увидите, как растет показатель Paged Pool (Выгружаемая память). То же самое можно увидеть в окне System Information утилиты Process Explorer. (Выберите Show и System Information.)
3. Чтобы определить метку пула, где происходит утечка, запустите Poolmon и нажмите клавишу *b*, чтобы сортировать по числу байтов. Дважды нажмите клавишу *p* для отображения в Poolmon только пула подкачиваемой памяти. Вы должны заметить, что пул с меткой «Leak» поднимается вверх по списку. (Poolmon выделяет строки, где происходят изменения.)
4. Теперь щелкните кнопку Stop Leaking, чтобы не истощить пул подкачиваемой памяти в своей системе.
5. Используя приемы, описанные в предыдущем разделе, запустите Strings (ее можно скачать с *www.sysinternals.com*) для поиска двоичных файлов драйвера, содержащих метку пула «Leak»:

```
Strings \windows\system32\drivers\*.sys | findstr Leak
```

Эта команда должна указать на файл Myfault.sys.

Ассоциативные списки

Windows поддерживает механизм быстрого выделения памяти — *ассоциативные списки* (look-aside lists). Главное различие между пулом и ассоциативным списком в том, что из пула можно выделять блоки памяти различного размера, а из ассоциативного списка — только фиксированные. Хотя пулы обеспечивают более высокую гибкость, ассоциативные списки работают быстрее, так как не используют спин-блокировку и не заставляют систему подбирать подходящую область свободной памяти, в которой мог бы уместиться текущий выделяемый блок.

Функции *ExInitializeNPagedLookasideList* и *ExInitializePagedLookasideList* (документированные в DDK) позволяют компонентам исполнительной системы и драйверам устройств создавать ассоциативные списки, размеры которых кратны размерам наиболее часто используемых структур данных. Для минимизации издержек, связанных с синхронизацией в многопроцессорных системах, некоторые компоненты исполнительной системы, в том числе диспетчер ввода-вывода, диспетчер кэша и диспетчер объектов, создают отдельные для каждого процессора ассоциативные списки, из которых выделяется память под часто используемые структуры данных. Сама исполнительная система создает для каждого процессора универсальные ассоциативные списки подкачиваемой и неподкачиваемой памяти с гранулярностью выделения в 256 байтов или менее.

Если ассоциативный список пуст (как это бывает сразу после его создания), система должна выделить память из подкачиваемого или неподкачиваемого пула. Но если в списке уже присутствует освобожденная структура, то занимаемая ею память выделяется очень быстро. (Список разрастается по мере возврата в него структур.) Процедуры выделения памяти из пула автоматически настраивают число освобожденных буферов, хранящихся в ассоциативном списке, в зависимости от частоты выделения памяти из этого списка драйвером или компонентом исполнительной системы. Чем чаще они выделяют память из списка, тем больше буферов в списке. Размер ассоциативных списков автоматически уменьшается, если память из них не выделяется. (Эта проверка выполняется раз в секунду, когда системный поток диспетчера настройки баланса пробуждается и вызывает функцию *KiAdjustLookasideDepth*.)

ЭКСПЕРИМЕНТ: просмотр системных ассоциативных списков

Содержимое и размер различных ассоциативных списков в системе можно просмотреть командой *!lookaside* отладчика ядра. Вот фрагмент вывода этой команды.

```
kd> !lookaside

Lookaside "nt!IopSmallIrpLookasideList" @ 804758a0 "Irps"
    Type     =     0000 NonPagedPool
```

```
    Current Depth  =        3   Max Depth  =        4
    Size           =      148   Max Alloc  =      592
    AllocateMisses =       32   FreeMisses =        9
    TotalAllocates =       52   TotalFrees =       32
    Hit Rate       =       38%  Hit Rate   =       71%
Lookaside "nt!IopLargeIrpLookasideList" @ 804756a0 "Irpl"
    Type     =     0000 NonPagedPool
    Current Depth  =        4   Max Depth  =        4
    Size           =      436   Max Alloc  =     1744
    AllocateMisses =     2623   FreeMisses =     2443
    TotalAllocates =     7039   TotalFrees =     6863
    Hit Rate       =       62%  Hit Rate   =       64%

Lookaside "nt!IopMdlLookasideList" @ 80475740 "Mdl "
    Type     =     0000 NonPagedPool
    Current Depth  =        3   Max Depth  =        4
    Size           =      120   Max Alloc  =      480
    AllocateMisses =     7017   FreeMisses =     1824
    TotalAllocates =    10901   TotalFrees =     5711
    Hit Rate       =       35%  Hit Rate   =       68%
...
Total NonPaged currently allocated for above lists = 4200
Total NonPaged potential for above lists           = 6144
Total Paged currently allocated for above lists    = 5136
Total Paged potential for above lists              = 12032
```

Утилита Driver Verifier

Driver Verifier представляет собой механизм, который можно использовать для поиска и локализации наиболее распространенных ошибок в драйверах устройств и другом системном коде режима ядра. Microsoft проверяет с помощью Driver Verifier свои драйверы и все драйверы, передаваемые производителями оборудования для тестирования на совместимость и включения в список Hardware Compatibility List (HCL). Такое тестирование гарантирует совместимость драйверов, включенных в список HCL, с Windows и отсутствие в них распространенных ошибок. (Существует и парная утилита Application Verifier, позволяющая улучшить качество кода пользовательского режима. Однако в этой книге она не рассматривается.)

Driver Verifier поддерживается несколькими системными компонентами — диспетчером памяти, диспетчером ввода-вывода и HAL, которые предусматривают параметры, включаемые для верификации драйверов. В этом разделе поясняются параметры верификации драйверов на отсутствие ошибок, связанных с управлением памятью (см. также главу 9).

Настройка и инициализация Driver Verifier

Для настройки Driver Verifier и просмотра статистики запустите Driver Verifier Manager (Диспетчер проверки драйверов), файл \Windows\System32\Veri-

fier.exe. После запуска появится окно с несколькими вкладками. Версия окна для Windows 2000 приведена на рис. 7-7. Чтобы указать, какие драйверы устройств вы хотите проверить, и задать типы проверок, используйте вкладку Settings (Параметры).

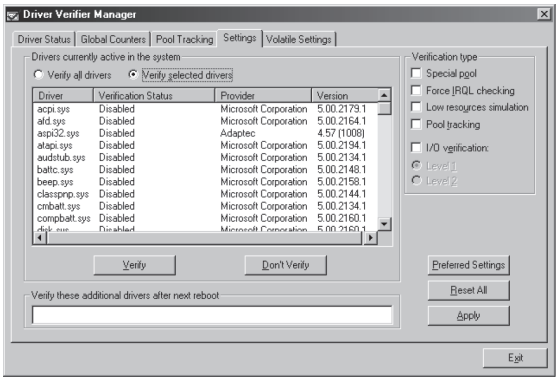


Рис. 7-7. *Driver Verifier Manager в Windows 2000*

В Windows XP и Windows Server 2003 этой утилите придали интерфейс в стиле мастера, как показано на рис. 7-8.

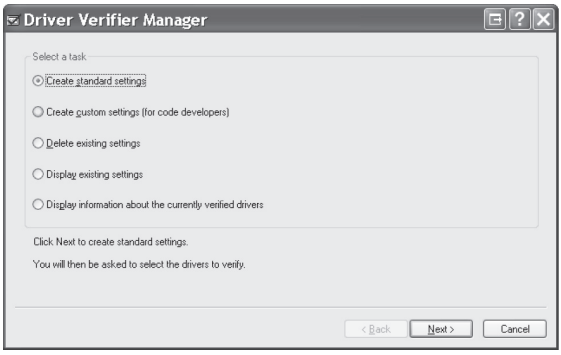


Рис. 7-8. *Driver Verifier Manager в Windows XP и Windows Server 2003*

Включать и отключать Driver Verifier, а также просматривать текущие параметры можно из командной строки этой утилиты. Для вывода списка ключей наберите **verifier /?**.

Настройки Driver Verifier Manager хранятся в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management. Параметр VerifyDriverLevel содержит битовую маску, представляющую включенные типы проверок. Имена проверяемых драйверов содержатся в параметре VerifyDrivers. (Эти параметры создаются в реестре только после выбора проверяемых драйверов в окне Driver Verifier Manager.) Если вы выберете верификацию всех драйверов, VerifyDrivers будет содержать символ звездочки. В зависимости от выбранных параметров может понадобиться перезагрузка системы.

На ранних этапах загрузки диспетчер памяти считывает из реестра значения этих параметров, определяя, какие драйверы следует верифицировать и какие параметры Driver Verifier включены. (Если загрузка происходит в безопасном режиме, все параметры Driver Verifier игнорируются.) Далее, если для проверки выбран хотя бы один драйвер, ядро сравнивает имя каждого загружаемого драйвера с именами драйверов, подлежащих верификации. Если имена совпадают, ядро вызывает функцию *MiApplyDriverVerifier*, которая заменяет все ссылки драйвера на функции ядра ссылками на эквивалентные функции Driver Verifier. Так, *ExAllocatePool* заменяется на *VerifierAllocatePool*. Драйвер подсистемы управления окнами производит аналогичные замены для использования эквивалентных функций Driver Verifier.

Теперь рассмотрим четыре параметра верификации драйверов, относящиеся к использованию памяти: Special Pool, Pool Tracking, Force IRQL Checking и Low Resources Simulation.

Special Pool (Особый пул) Этот параметр заставляет функции, отвечающие за выделение памяти из пулов, окружать выделяемый блок недействительными страницами, чтобы ссылки за пределы этого блока вызывали нарушение доступа в режиме ядра и последующий крах системы. А это позволяет тут же указать пальцем на сбойный драйвер. Параметр Special Pool также заставляет проводить дополнительные проверки, когда драйвер выделяет или освобождает память.

При включении параметра Special Pool функции пулов выделяют в памяти ядра регион для Driver Verifier, и последний перенаправляет запросы проверяемого драйвера на выделение памяти в особый пул, а не в стандартные пулы. При выделении драйвером памяти из особого пула Driver Verifier округляет размер выделяемого блока до размера, кратного размеру страницы. Поскольку Driver Verifier окружает выделенный блок недействительными страницами, при попытке записи или чтения за пределами этого блока драйвер попадает на недействительную страницу, и диспетчер памяти сообщает о нарушении доступа в режиме ядра.

На рис. 7-9 приведен пример блока, выделенного Driver Verifier в особом пуле для проверяемого драйвера устройства.

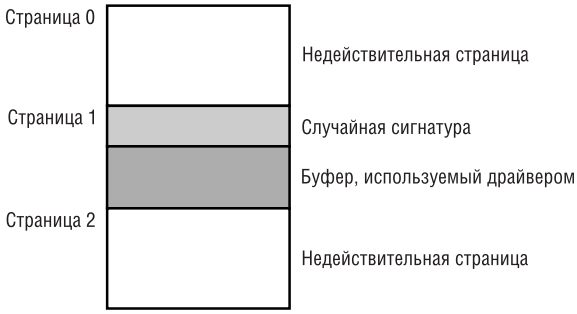


Рис. 7-9. *Структура области памяти, выделяемой в особом пуле*

По умолчанию Driver Verifier распознает ошибки, связанные с попытками обращения за верхнюю границу выделенного блока (overrun errors). Он делает это, помещая используемый драйвером буфер в конец выделенной страницы и заполняя ее начало случайными значениями. Хотя Driver Verifier Manager не предусматривает параметр для включения детекции ошибок, связанных с попытками обращения за нижнюю границу выделенного блока (underrun errors), вы можете активизировать ее вручную, добавив в раздел реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory-Management параметр PoolTagOverruns типа DWORD и присвоив ему значение 0 (или запустив утилиту Gflags и установив флажок Verify Start вместо установленного по умолчанию Verify End). Тогда Driver Verifier будет размещать буфер драйвера не в конце, а в начале страницы.

Конфигурация, при которой Driver Verifier обнаруживает ошибки типа «overrun», до некоторой степени обеспечивает и распознавание ошибок типа «underrun». Когда драйвер освобождает буфер и возвращает его в Driver Verifier, последний должен убедиться, что содержимое памяти, предшествующее буферу, не изменилось. Иное означает, что драйвер обратился к памяти, расположенной до начала буфера, и что-то записал за пределами этого буфера.

При выделении памяти из особого пула и ее освобождении также проверяется корректность IRQL процессора. Эта проверка позволяет выявить ошибку, встречающуюся в некоторых драйверах, из-за которой они пытаются выделять память в подкачиваемом пуле при IRQL уровня «DPC/dispatch» или выше.

Особый пул можно сконфигурировать и вручную, добавив в раздел реестра HKLM\SYSTEMCurrentControlSet\Control\Session Manager\Memory Management параметр PoolTag типа DWORD; он представляет тэги выделенной памяти, используемые системой для особого пула. Тогда, даже если Driver Verifier не настроен на верификацию данного драйвера, при совпадении тэга, сопоставленного с выделенной драйвером памятью, и значения PoolTag, память будет выделяться из особого пула. Если вы присвоите PoolTag значение 0x0000002a или символ подстановки (*), то при наличии достаточного количества физической и виртуальной памяти вся память для драйверов будет выделяться из особого пула. (Если памяти не хватит, драйверы вернутся к использованию обычного пула; размер каждого выделяемого блока ограничен двумя страницами.)

Pool Tracking (Слежение за пулом) Если параметр Pool Tracking активен, диспетчер памяти проверяет при выгрузке драйвера, освободил ли тот всю выделенную для него память. Если нет, диспетчер памяти вызывает крах системы и сообщает о сбойном драйвере. Driver Verifier тоже показывает общую статистику по использованию пула — откройте вкладку Pool Tracking (Слежение за пулом) в Driver Verifier Manager (Диспетчер проверки драйверов). Кроме того, пригодится и команда *!verifier* отладчика ядра; она, кстати, выводит больше информации, чем Driver Verifier.

Force IRQL Checking (Обяз. проверка IRQL) Одна из самых распространенных ошибок в драйверах устройств — попытка обращения к стра-

ничному файлу при слишком высоком уровне IRQL процессора. Как уже говорилось в главе 3, диспетчер памяти не обрабатывает ошибки страниц при IRQL уровня «DPC/dispatch» или выше. Система часто не распознает экземпляры драйвера, обращающиеся к данным из подкачиваемого пула при повышенном IRQL процессора, поскольку в этот момент такие данные физически присутствуют в памяти. Но в других случаях, если эти данные выгружены в страничный файл, попытка обращения к ним вызывает крах системы со стоп-кодом IRQL_NOT_LESS_OR_EQUAL (т. е. IRQL превышает тот уровень, при котором возможно обращение к подкачиваемой памяти).

Проверка драйверов устройств на наличие подобной ошибки — дело очень трудное, но Driver Verifier упрощает эту задачу. Если параметр Force IRQL Checking включен, Driver Verifier выводит весь подкачиваемый код и данные режима ядра из системного рабочего набора всякий раз, когда проверяемый драйвер повышает IRQL. Это делается с помощью внутренней функции *MmTrimAllSystemPagableMemory*. При любой попытке проверяемого драйвера обратиться к подкачиваемой памяти при повышенном IRQL система фиксирует нарушение доступа и происходит крах с сообщением, указывающим на сбойный драйвер.

Low Resources Simulation (Нехватка ресурсов) При включении этого параметра Driver Verifier случайным образом отклоняет некоторые запросы драйвера на выделение памяти. Раньше разработчики создавали многие драйверы устройств в расчете на то, что памяти ядра всегда достаточно, так как иное означало бы, что система все равно вот-вот рухнет. Но, поскольку временная нехватка памяти иногда возможна, драйверы устройств должны корректно обрабатывать ошибки выделения памяти при ее нехватке.

Через 7 минут после загрузки системы (этого времени достаточно для завершения критического периода инициализации, когда из-за нехватки памяти драйвер мог бы просто не загрузиться) Driver Verifier начинает случайным образом отклонять запросы проверяемых драйверов на выделение памяти. Если драйвер не в состоянии корректно обработать ошибки выделения памяти, это скорее всего проявится в виде краха системы.

Driver Verifier представляет собой ценное пополнение в арсенале средств верификации и отладки, доступном разработчикам драйверов устройств. Этот инструмент позволил с ходу выявить ошибки во многих драйверах. Так что Driver Verifier тоже внес вклад в повышение качества кода Windows, работающего в режиме ядра.

Структуры виртуального адресного пространства

Здесь описываются компоненты в пользовательском и системном адресных пространствах, а также специфика адресных пространств в 32- и 64-разрядных системах. Эта информация поможет вам понять ограничения на виртуальную память для процессов и системы на обеих платформах.

На виртуальное адресное пространство в Windows проецируются три основных вида данных: код и данные, принадлежащие процессу, код и данные, принадлежащие сеансу, а также общесистемные код и данные.

Как мы поясняли в главе 1, каждому процессу выделяется собственное адресное пространство, недоступное другим процессам (если только у них нет разрешения на открытие процесса с правами доступа для чтения и записи). Потоки внутри процесса никогда не получают доступа к виртуальным адресам вне адресного пространства своего процесса, если только не проецируют данные на раздел общей памяти и/или не используют специальные функции, позволяющие обращаться к адресному пространству другого процесса. Сведения о виртуальном адресном пространстве процесса хранятся в *таблицах страниц* (page tables), которые рассматриваются в разделе по трансляции адресов. Таблицы страниц размещаются на страницах памяти, доступных только в режиме ядра, поэтому пользовательские потоки в процессе не могут модифицировать структуру адресного пространства своего процесса.

В системах с поддержкой нескольких сеансов (Windows 2000 Server с установленной службой Terminal Services, Windows XP и Windows Server 2003) пространство сеанса содержит информацию, глобальную для каждого сеанса. (Подробное описание сеанса см. в главе 2.) *Сеанс* (session) состоит из процессов и других системных объектов (вроде WindowStation, рабочих столов и окон). Эти объекты представляют сеанс единственного пользователя, который зарегистрировался на рабочей станции. У каждого сеанса есть своя область пула подкачиваемой памяти, используемая подсистемой Windows (Win32k.sys) для выделения памяти под сеансовые GUI-структуры данных. Кроме того, каждый сеанс получает свою копию процесса подсистемы Windows (Csrss.exe) и Winlogon.exe. За создание новых сеансов отвечает процесс диспетчера сеансов (Smss.exe). Его задачи включают загрузку сеансовых копий Win32k.sys и создание специфических для сеанса экземпляров процессов Csrss и Winlogon, а также пространства имен диспетчера объектов.

Для виртуализации сеансов все общие для сеанса структуры данных проецируются на область системного пространства, которая называется *пространством сеанса* (session space). При создании процесса этот диапазон адресов проецируется на страницы, принадлежащие тому сеансу, к которому относится данный процесс. Размер области для проецируемых представлений в пространстве сеанса можно настраивать, используя параметры в разделе реестра HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management. (В 32-разрядных системах эти параметры игнорируются при загрузке системы с параметром /3GB.)

Наконец, *системное пространство* содержит глобальные код и структуры данных операционной системы, видимые каждому процессу. Системное пространство состоит из следующих компонентов:

- **Системный код** Содержит образ операционной системы, HAL и драйверы устройств, используемые для загрузки системы.

- **Представления, проецируемые системой** Сюда проецируются Win32k.sys, загружаемая часть подсистемы Windows режима ядра, а также используемые ею графические драйверы режима ядра (подробнее о Win32k.sys см. главу 2).
- **Гиперпространство** Особая область, применяемая для проецирования списка рабочего набора процесса и временного проецирования других физических страниц для таких операций, как обнуление страницы из списка свободных страниц (если список обнуленных страниц пуст и нужна обнуленная страница), подготовка адресного пространства при создании нового процесса и объявление недействительными PTE в других таблицах страниц (например, при удалении страницы из списка простаивающих страниц).
- **Список системного рабочего набора** Структуры данных списка рабочего набора, описывающие системный рабочий набор.
- **Системный кэш** Виртуальное адресное пространство, применяемое для проецирования файлов, открытых в системном кэше. (О диспетчере кэша см. главу 11.)
- **Пул подкачиваемой памяти** Системная куча подкачиваемой памяти.
- **Элементы системной таблицы страниц (PTE)** Пул системных PTE, используемых для проецирования таких системных страниц, как пространство ввода-вывода, стеки ядра и списки дескрипторов памяти. Вы можете узнать, сколько системных PTE доступно, проверив значение счетчика Memory: Free System Page Table Entries (Память: Свободных элементов таблицы страниц) в оснастке Performance (Производительность).
- **Пул неподкачиваемой памяти** Системная куча неподкачиваемой памяти, обычно состоящая из двух частей, которые располагаются внизу и вверху системного пространства.
- **Данные аварийного дампа** Область, зарезервированная для записи информации о состоянии системы на момент краха.
- **Область, используемая HAL** Область, зарезервированная под структуры, специфичные для HAL.

ПРИМЕЧАНИЕ Внутреннее название системного рабочего набора — рабочий набор системного кэша (system cache working set). Однако этот термин неудачен, так как в системный рабочий набор входит не только кэш, но и пул подкачиваемой памяти, подкачиваемые системные код и данные, а также подкачиваемые код и данные драйверов.

Теперь после краткого обзора базовых компонентов виртуального адресного пространства в Windows давайте рассмотрим специфику структур этого пространства на платформах x86, IA64 и x64.

Структуры пользовательского адресного пространства на платформе x86

По умолчанию каждый пользовательский процесс в 32-разрядной версии Windows располагает собственным адресным пространством размером до Гб; остальные 2 Гб забирает себе операционная система. Windows 2000 Advanced Server, Windows 2000 Datacenter Server, Windows XP Service Pack 2 и выше, а также Windows Server 2003 (все версии) поддерживают загрузочный параметр (ключ /3GB в Boot.ini), позволяющий создавать пользовательские адресные пространства размером по 3 Гб. Windows XP и Windows Server 2003 поддерживают дополнительный ключ (/USERVA), который дает возможность задавать размер пользовательского адресного пространства между 2 и 3 Гб (значение указывается в мегабайтах). Структуры этих двух адресных пространств показаны на рис. 7-10.

Поддержка возможности расширения пользовательского адресного пространства для 32-разрядного процесса за пределы 2 Гб введена как временное решение для поддержки приложений вроде серверов баз данных, которым для хранения данных требуется больше памяти, чем возможно в 2-гигабайтном адресном пространстве. Но лучше, конечно, пользоваться уже рассмотренными AWE-функциями.

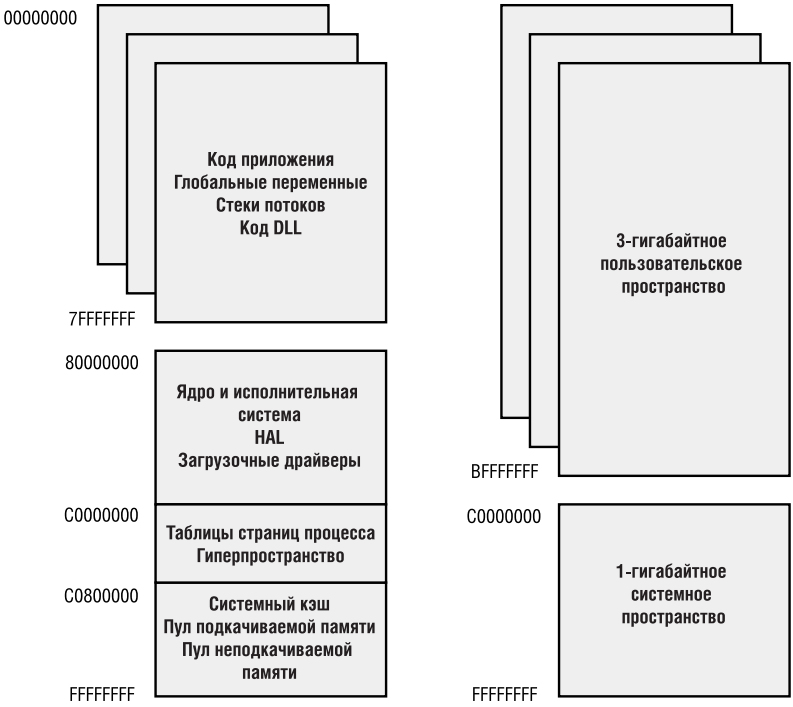


Рис. 7-10. *Структуры виртуального адресного пространства в системах типа x86*

Для расширения адресного пространства процесса за пределы 2 Гб в заголовке образа должен быть указан флаг IMAGE_FILE_LARGE_ADDRESS_AWARE. Иначе Windows резервирует это дополнительное пространство, и виртуальные адреса выше 0x7FFFFFFF становятся недоступны приложению. (Так делается, чтобы избежать краха приложения, не способного работать с этими адресами.) Этот флаг можно задать ключом компоновщика /LARGEADDRESSAWARE при сборке исполняемого файла. Данный флаг не действует при запуске приложения в системе с 2-гигабайтным адресным пространством для пользовательских процессов. (Если вы загрузите любую версию Windows Server с параметром /3GB, размер системного пространства уменьшится до 1 Гб, но пользовательское пространство все равно останется двухгигабайтным, даже несмотря на поддержку запускаемой программой большого адресного пространства.)

Несколько системных образов помечаются как поддерживающие большие адресные пространства, благодаря чему они могут использовать преимущества систем, работающих с такими пространствами. К их числу относятся:

- Lsass.exe — подсистема локальной аутентификации;
- Inetinfo.exe — Internet Information Services (IIS);
- Chkdsk.exe — утилита Check Disk;
- Dllhst3g.exe — специальная версия Dllhost.exe (для COM+-приложений).

Наконец, поскольку по умолчанию память, выделяемая через *VirtualAlloc*, начинается с младших адресов (если только процесс не выделяет очень много виртуальной памяти или не имеет очень сильно фрагментированного виртуального адресного пространства), она никогда не достигает самых старших адресов. Поэтому при тестировании вы можете указать, что выделение памяти должно начинаться со старших адресов. Для этого добавьте в реестр DWORD-параметр HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management\AllocationPreference и присвойте ему значение 0x100000.

Структура системного адресного пространства на платформе x86

В этом разделе подробно описывается структура и содержимое системного пространства в 32-разрядной Windows. На рис. 7-11 показана общая схема 2-гигабайтного системного пространства на платформе x86.

В таблице 7-8 перечислены переменные ядра, содержащие стартовые и конечные адреса различных регионов системного пространства: одни из них фиксированы, а другие вычисляются при загрузке с учетом доступного объема системной памяти и выпуска операционной системы Windows — клиентского или серверного.

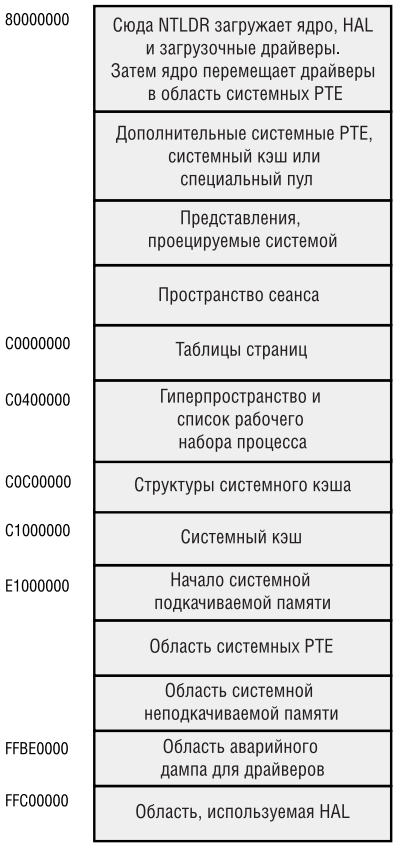


Рис. 7-11. *Структура системного пространства в x86-системах (пропорции не соблюдены)*

Таблица 7-8. *Системные переменные, определяющие адреса регионов в системном пространстве*

Системная переменная	Описание	2-гигабайтное системное пространство в x86-системах (без PAE)	1-гигабайтное системное пространство в x86-системах (без PAE)
MmSystemRangeStart	Стартовый адрес системного пространства	0x80000000	0xC0000000
MmSystemCacheWorkingSetList	Список системного рабочего набора	0xC0C00000	Вычисляется
MmSystemCacheStart	Начало системного кэша	Вычисляется	Вычисляется
MmSystemCacheEnd	Конец системного кэша	Вычисляется	Вычисляется
MiSystemCacheStartExtra	Начало системного кэша или области расширения системных PTE	Вычисляется	0

Таблица 7-8. (*окончание*)

Системная переменная	Описание	2-гигабайтное системное пространство в x86-системах (без PAE)	1-гигабайтное системное пространство в x86-системах (без PAE)
MiSystemCacheEndExtra	Конец системного кэша или области расширения системных PTE	0xC0000000	0
MmPagedPoolStart	Начало подкачиваемого пула	Вычисляется	Вычисляется
MmPagedPoolEnd	Конец подкачиваемого пула	Вычисляется (максимум 650 Мб)	Вычисляется (минимум 160 Мб)
MmNonPagedSystemStart	Начало системных PTE	Вычисляется (минимум — 0xEB000000)	Вычисляется
MmNonPagedPoolStart	Начало неподкачиваемого пула	Вычисляется	Вычисляется
MmNonPagedPoolExpansionStart	Начало области расширения неподкачиваемого пула	Вычисляется	Вычисляется
MmNonPagedPoolEnd	Конец неподкачиваемого пула	0xFFBE0000	0xFFBE0000

Пространство сеанса на платформе x86

В системах с поддержкой нескольких сеансов код и данные, уникальные для каждого сеанса, проецируются в системное адресное пространство, но разделяются всеми процессами в данном сеансе. Общая схема сеансового пространства представлена на рис. 7-12.

Рис. 7-12. *Структура сеансового пространства в x86-системах (пропорции не соблюдены)*

Размеры областей в сеансовом пространстве можно настраивать, добавляя параметры в раздел реестра HKLM\System\CurrentControlSet\Control\Session

Manager\Memory Management. Эти параметры и соответствующие переменные ядра, которые содержат реальные значения, перечислены в таблице 7-9.

Таблица 7-9. *Параметры конфигурации сеансового пространства*

Параметры реестра	Переменные ядра
SessionImageSize	*MmSessionImageSize*
SessionViewSize	*MmSessionViewSize*
SessionPoolSize	*MmSessionPoolSize*

ЭКСПЕРИМЕНТ: просмотр сеансов

Узнать, какие процессы и к каким сеансам относятся, можно по счетчику производительности Session ID (Код сеанса). Он доступен через диспетчер задач, Process Explorer или оснастку Performance (Производительность). Используя команду *!session* отладчика ядра, можно перечислить активные сеансы:

```
lkd> !session
Sessions on machine: 3
Valid Sessions: 0 1 2
Current Session 0
```

Далее вы можете установить активный сеанс командой *!session –s* и вывести адрес сеансовых структур данных и список процессов в этом сеансе командой *!sprocess*:

```
lkd> !session -s 2
Sessions on machine: 3
Implicit process is now 8144f3a0
Using session 2
lkd> !sprocess
Dumping Session 2

_MM_SESSION_SPACE f9d5a000
_MMSESSION        f9d5ad80
PROCESS 8144f3a0  SessionId: 2  Cid: 0d18    Peb: 7ffdf000
    ParentCid: 0134  DirBase: 07ef7000  ObjectTable: e1855bb0
    HandleCount:  71.  Image: csrss.exe

PROCESS 813f6500  SessionId: 2  Cid: 0d04    Peb: 7ffdf000
    ParentCid: 0134  DirBase: 0f3fd000  ObjectTable: e18f3008
    HandleCount: 131.  Image: winlogon.exe
```

Для просмотра детальных сведений о сеансе выведите дамп структуры MM_SESSION_SPACE командой *dt*:

```
lkd> dt nt!_mm_session_space f9d4a000
nt!_MM_SESSION_SPACE
   +0x000 GlobalVirtualAddress : 0xf9d4a000
   +0x004 ReferenceCount   : 0x19
   +0x008 u                : __unnamed
```

```
+0x00c SessionId        : 0
+0x010 ProcessList      : _LIST_ENTRY [ 0x817d92cc - 0x812de88c ]
+0x018 LastProcessSwappedOutTime : _LARGE_INTEGER 0x0
+0x020 SessionPageDirectoryIndex : 0x6a29
+0x024 NonPagablePages  : 0xf
+0x028 CommittedPages   : 0x3f
+0x02c PagedPoolStart   : 0xbc000000
+0x030 PagedPoolEnd     : 0xbc3fffff
+0x034 PagedPoolBasePde : 0xc0300bc0
+0x038 Color            : 0xd6
+0x03c ProcessOutSwapCount : 5
+0x040 SessionPoolAllocationFailures : [4] 0
+0x050 ImageList        : _LIST_ENTRY [ 0x81801fa8 - 0x816933a0 ]
+0x058 LocaleId         : 0x409
+0x05c AttachCount      : 0
+0x060 AttachEvent      : _KEVENT
+0x070 LastProcess      : (null)
+0x074 ProcessReferenceToSession : 25
+0x078 WsListEntry      : _LIST_ENTRY [ 0x8056d058 - 0x8056d058 ]
+0x080 Lookaside        : [26] _GENERAL_LOOKASIDE
+0xd80 Session          : _MMSESSION
+0xdc0 PagedPoolMutex   : _KGUARDED_MUTEX
+0xde0 PagedPoolInfo    : _MM_PAGED_POOL_INFO
+0xe00 Vm               : _MMSUPPORT
+0xe60 Wsle             : 0xbf400038
+0xe64 Win32KDriverUnload : 0xbf9169ab     +ffffffffbf9169ab
+0xe68 PagedPool        : _POOL_DESCRIPTOR
+0x1e98 PageTables       : 0x816e7008
+0x1e9c ImageLoadingCount : 0
```

ЭКСПЕРИМЕНТ: просмотр памяти, используемой пространством сеанса

Просмотреть, как используется память в пространстве сеанса, позволяет команда *!vm 4* отладчика ядра. Вот пример для 32-разрядной системы Windows Server 2003 Enterprise Edition с двумя активными сеансами:

```
lkd> !vm 4
...
Terminal Server Memory Usage By Session:

Session Paged Pool Maximum is 4096K
Session View Space Maximum is 49152K

Session ID 0 @ f9d4a000:
Paged Pool Usage:           0K
Commit Usage:             252K

Session ID 1 @ f9d4c000:
Paged Pool Usage:           0K
Commit Usage:             172K
```

см. след. стр.

Та же команда применительно к 64-разрядной системе Windows Server 2003 Enterprise Edition с двумя активными сеансами дает следующий вывод:

```
Terminal Server Memory Usage By Session:
Session Paged Pool Maximum is 65536K
Session View Space Maximum is 106496K

Session ID 0 @ fffffadfe4bb9000:
Paged Pool Usage:           0K
Commit Usage:            4608K
Session ID 1 @ fffffadfe4c5f000:
Paged Pool Usage:           0K
Commit Usage:             584K
```

Системные PTE

Системные PTE используются для динамического проецирования системных страниц, в частности пространства ввода-вывода, стеков ядра и списков дескрипторов памяти. Системные PTE не являются неисчерпаемым ресурсом. Например, Windows 2000 может описывать всего 660 Мб системного виртуального адресного пространства (из которых 440 Мб могут быть непрерывными). В 32-разрядных версиях Windows XP и Windows Server 2003 число доступных системных PTE увеличилось, благодаря чему система может описывать до 1,3 Гб системного виртуального адресного пространства, из которых 960 Мб могут быть непрерывными. В 64-разрядной Windows системные PTE позволяют описывать до 128 Гб непрерывного виртуального адресного пространства.

Число системных PTE показывается счетчиком Memory: Free System Page Table Entries (Память: Свободных элементов таблицы страниц) в оснастке Performance. По умолчанию Windows при загрузке подсчитывает, сколько системных PTE нужно создать, исходя из объема доступной памяти. Чтобы изменить это число, присвойте параметру реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\SystemPages значение, равное нужному вам количеству PTE. (Это может понадобиться для поддержки устройств, требующих большого количества системных PTE, например видеоплат с 512 Мб видеопамяти, которую нужно спроецировать всю сразу.) Если параметр содержит значение 0xFFFFFFFF, резервируется максимальное число системных PTE.

Структуры 64-разрядных адресных пространств

Теоретически 64-разрядное виртуальное адресное пространство может быть до 16 экзабайтов (18 446 744 073 709 551 616 байтов, или примерно 17,2 миллиарда гигабайтов). В отличие от 32-разрядного адресного пространства на платформ x86, где по умолчанию оно делится на две равные части (половина для процесса и половина для системы), 64-разрядное адресное пространство делится на ряд регионов разного размера, компоненты которого концеп-

туально совпадают с порциями пользовательского, системного и сеансового пространств. Размер этих регионов (таблица 7-10) отражает лимиты текущей реализации, которые могут быть расширены в будущих выпусках.

Детальные структуры адресных пространств IA64 и x64 различаются незначительно. Структуру адресного пространства для IA64 см. на рис. 7-13, а для x64 — на рис. 7-14.

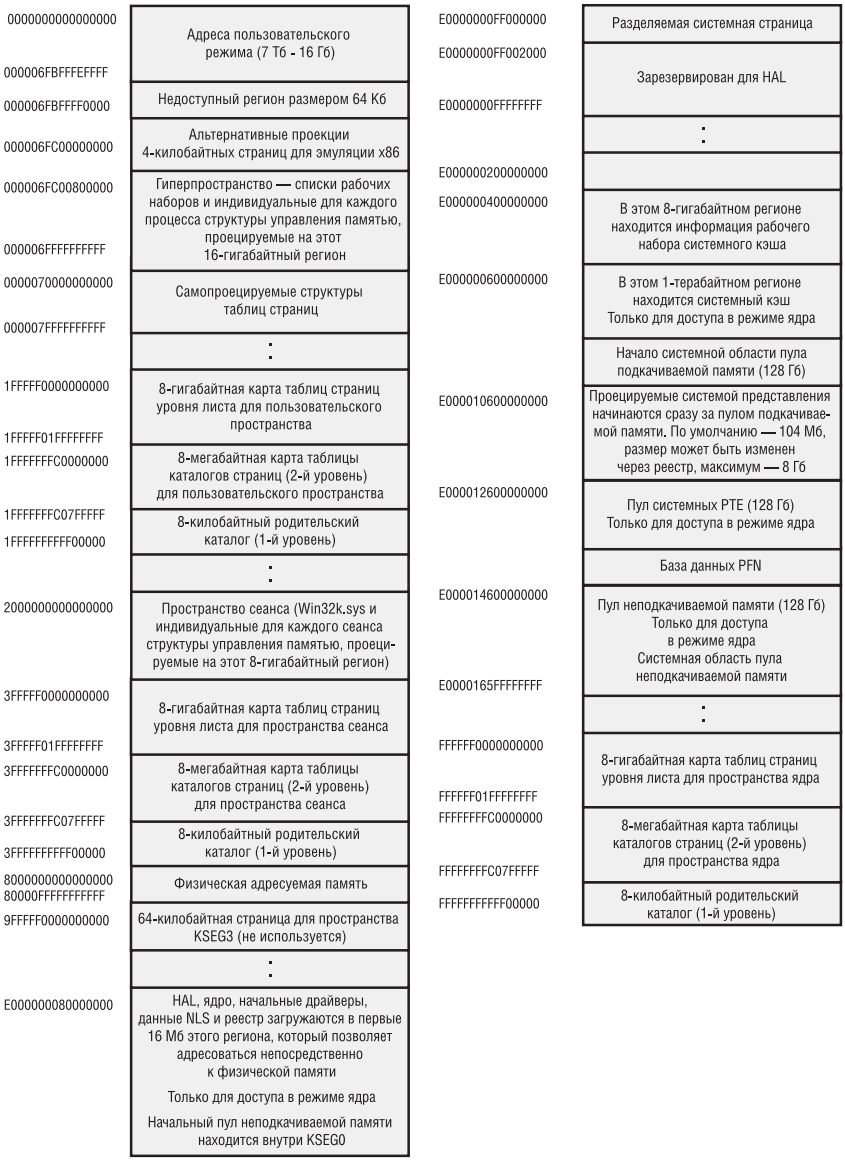


Рис. 7-13. *Структура адресного пространства на платформе IA64*

Таблица 7-10. *Размеры регионов в 64-разрядном адресном пространстве*

Регион	IA64	x64	x86
Адресное пространство процесса	7152 Гб	8192 Гб	2–3 Гб
Пространство системных PTE	128 Гб	128 Гб	1,2 Гб
Системный кэш	128 Гб	128 Гб	960 Мб
Пул подкачиваемой памяти	128 Гб	128 Гб	470–650 Мб
Пул неподкачиваемой памяти	128 Гб	128 Гб	256 Мб

Рис. 7-14. *Структура адресного пространства на платформе x64*

Трансляция адресов

Теперь, когда вы познакомились со структурами виртуального адресного пространства в Windows, рассмотрим, как она увязывает эти адресные пространства со страницами физической памяти (приложения и системный код используют виртуальные адреса). Мы начнем с детального описания трансляции 32-разрядных адресов на платформе x86, потом кратко поясним ее отличия на 64-разрядных платформах IA64 и x64. В следующем разделе вы узнаете, что происходит, когда виртуальный адрес не удается разрешить в физический (из-за выгрузки в страничный файл), и как Windows управляет физической памятью через рабочие наборы и базу данных номеров фреймов страниц.

Трансляция виртуальных адресов на платформе x86

С помощью структур данных (таблиц страниц), создаваемых и поддерживаемых диспетчером памяти, процессор транслирует виртуальные адреса в физические. Каждый виртуальный адрес сопоставлен со структурой системного пространства, которая называется *элементом таблицы страниц* (page table entry, PTE) и содержит физический адрес, соответствующий виртуальному. Например, на рис. 7-15 показаны три последовательно расположенные виртуальные страницы, проецируемые на три разрозненные физические страницы (платформа x86).

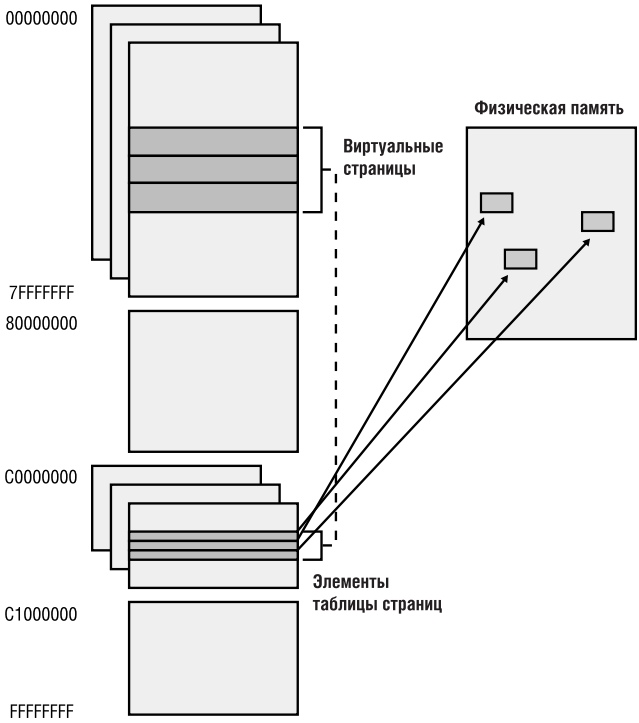


Рис. 7-15. *Трансляция виртуальных адресов в физические (x86)*

Пунктирные линии на рис. 7-15 соединяют виртуальные страницы с PTE, представляя косвенные связи между виртуальными и физическими страницами.

ПРИМЕЧАНИЕ Код режима ядра (например, драйверов устройств) может ссылаться на физические адреса, транслируя их в виртуальные. Подробнее об этом см. описание функций поддержки списка дескрипторов памяти (memory descriptor list, MDL) в DDK.

По умолчанию в x86-системе Windows для трансляции виртуальных адресов в физические использует двухуровневую таблицу страниц (x86-системы, работающие с PAE-версией ядра, используют трехуровневую таблицу страниц, но они в этом разделе не рассматриваются). 32-разрядный виртуальный адрес интерпретируется как совокупность трех элементов: индекса каталога страниц, индекса таблицы страниц и индекса байта. Они применяются в качестве указателей в структурах, описывающих проекции страниц (рис. 7-16). Размеры страницы и PTE определяет размеры каталога страниц и полей индекса таблицы страниц. Так, в x86-системах длина индекса байта составляет 12 битов, поскольку размер страницы равен 4096 байтов (т. е. 2^{12}).

Рис. 7-16. *Элементы 32-разрядного виртуального адреса в x86-системах*

Индекс каталога страниц (page directory index) применяется для поиска таблицы страниц, содержащей PTE для данного виртуального адреса. С помощью *индекса таблицы страниц* (page table index) осуществляется поиск PTE, который, как уже говорилось, содержит физический адрес, по которому проецируется виртуальная страница. *Индекс байта* (byte index) позволяет найти конкретный адрес на физической странице. Взаимосвязи этих трех величин и их использование для трансляции виртуальных адресов в физические показаны на рис. 7-17.

При трансляции виртуального адреса выполняются следующие операции.

1. Аппаратные средства управления памятью находят каталог страниц текущего процесса. При каждом переключении контекста процесса эти средства получают адрес каталога страниц нового процесса. Обычно операционная система записывает этот адрес в специальный регистр процессора.
2. Индекс каталога страниц используется как указатель для поиска элемента каталога страниц (page directory entry, PDE), который определяет местонахождение таблицы страниц, нужной для трансляции виртуального адреса. PDE содержит номер фрейма страницы (page frame number, PFN) таблицы страниц (если она находится в памяти; однако такие таблицы могут выгружаться в страничный файл).

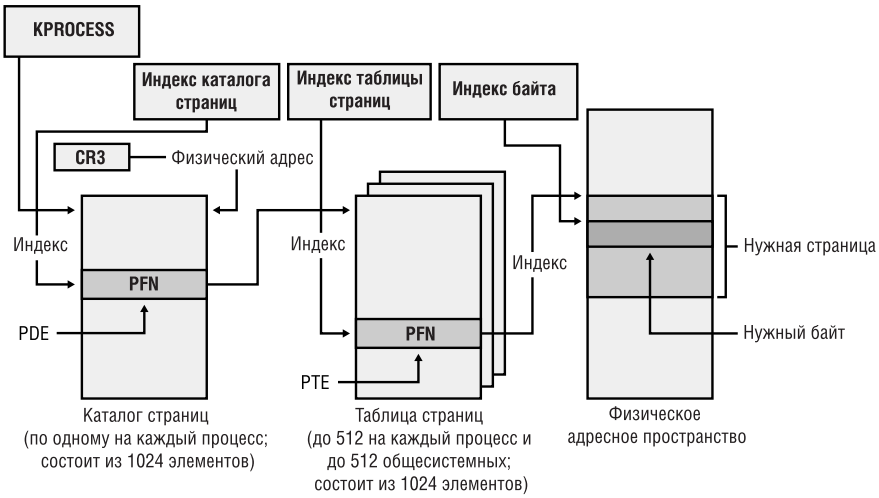


Рис. 7-17. *Трансляция виртуального адреса в x86-системах*

3. Индекс таблицы страниц используется как указатель для поиска PTE, который определяет местонахождение требуемой виртуальной страницы.
4. На основе PTE отыскивается страница. Если она действительна, то содержит PFN соответствующей страницы физической памяти. Если PTE сообщает, что страница недействительна, обработчик ошибок подсистемы управления памятью пытается найти страницу и сделать ее действительной (см. раздел по обработке ошибок страниц далее в этой главе). Если сделать страницу действительной не удалось (например, из-за ошибки защиты), обработчик ошибок генерирует нарушение доступа или вызывает переход в состояние отладки.
5. Если PTE указывает на действительную страницу, для поиска адреса нужных данных на физической странице используется индекс байта.

Ознакомившись с общей картиной, перейдем к детальному рассмотрению структуры каталогов страниц, таблиц страниц и PTE.

Каталоги страниц

У каждого процесса есть один *каталог страниц* (page directory), который представляет собой страницу с адресами всех таблиц страниц для данного процесса. Физический адрес каталога страниц процесса хранится в блоке KPROCESS и проецируется по адресу 0xC0300000 в x86-системах или 0xC0600000 в системах с PAE-ядром. Весь код, выполняемый в режиме ядра, использует не физические, а виртуальные адреса (о KPROCESS см. главу 6).

Процессору известно местонахождение страницы каталога страниц, поскольку в специальный регистр процессора (CR3 в x86-системах) загружен ее физический адрес. При каждом переключении контекста на поток другого процесса процедура ядра, отвечающая за переключение контекста, загружает в этот регистр значение из блока KPROCESS нового процесса. Переключе-

ние контекста между потоками одного процесса не влечет перезагрузку физического адреса каталога страниц, поскольку все потоки одного процесса используют одно и то же адресное пространство.

Каталог страниц состоит из элементов (PDE), каждый из которых имеет длину 4 байта (в системах с PAE-ядром — 8 байтов) и описывает состояние и адреса всех возможных таблиц страниц для данного процесса. (Как будет сказано далее, таблицы страниц создаются по мере необходимости, так что каталоги страниц большинства процессов ссылаются лишь на небольшой набор таких таблиц.) Мы не будем отдельно рассматривать формат PDE, поскольку он в основном совпадает с форматом аппаратного PTE.

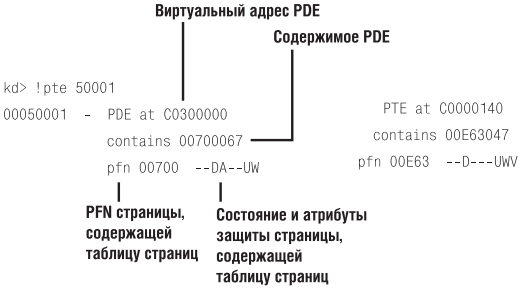
В x86-системах без PAE для полного описания 4-гигабайтного виртуального адресного пространства требуется 1024 таблицы страниц. Каталог страниц процесса, связывающий эти таблицы, содержит 1024 PDE. Соответственно размер индекса каталога равен 10 битам ($2^{10} = 1024$). В x86-системах, работающих в режиме PAE, в таблице страниц 512 элементов (размер индекса каталога страниц равен 9 битам). Из-за наличия 4 каталогов страниц максимальное число таблиц страниц составляет 2048.

ЭКСПЕРИМЕНТ: исследуем каталог страниц и PDE

Физический адрес каталога страниц текущего процесса можно увидеть, изучив поле DirBase в выходной информации команды *!process* отладчика ядра.

```
kd> !process
PROCESS 80145880  Cid: 0000   Peb: 00000000   ParentCid: 0000
    DirBase: 00030000  ObjectTable: 80695668  TableSize: 126.
    Image: Idle
    VadRoot 0 Clone 0 Private 0. Modified 0. Locked 0.
    80145A3C MutantState Locked OwningThread 0
    Process Lock Owned by Thread 0
```

Виртуальный адрес каталога страниц можно выяснить из информации, выводимой отладчиком ядра для PTE конкретного виртуального адреса, как показано ниже.



Часть выходной информации отладчика ядра, касающаяся PTE, рассматривается в разделе «Страницы таблиц и PTE» далее в этой главе.

Поскольку Windows предоставляет каждому процессу закрытое адресное пространство, у каждого процесса свой набор таблиц страниц для проецирования этого пространства. В то же время таблицы страниц, описывающие системное пространство, разделяются всеми процессами (а пространство сеанса разделяется процессами в сеансе). Чтобы не допустить появления нескольких таблиц страниц, описывающих одну и ту же виртуальную память, при создании процесса PDE, описывающие системное пространство, инициализируются так, чтобы они указывали на существующие системные таблицы страниц. Если процесс является частью сеанса, таблицы страниц, описывающие пространство сеанса, тоже используются совместно. Для этого PDE пространства сеанса инициализируются так, чтобы они указывали на существующие сеансовые таблицы страниц.

Однако, как показано на рис. 7-18, не все процессы имеют одинаковое представление системного пространства. Так, если при расширении пула подкачиваемой памяти требуется создать новую системную таблицу страниц, диспетчер памяти — вместо того чтобы сразу записывать указатели на новую системную таблицу во все каталоги страниц процессов — обновляет эти каталоги только по мере обращения процессов по новому виртуальному адресу.

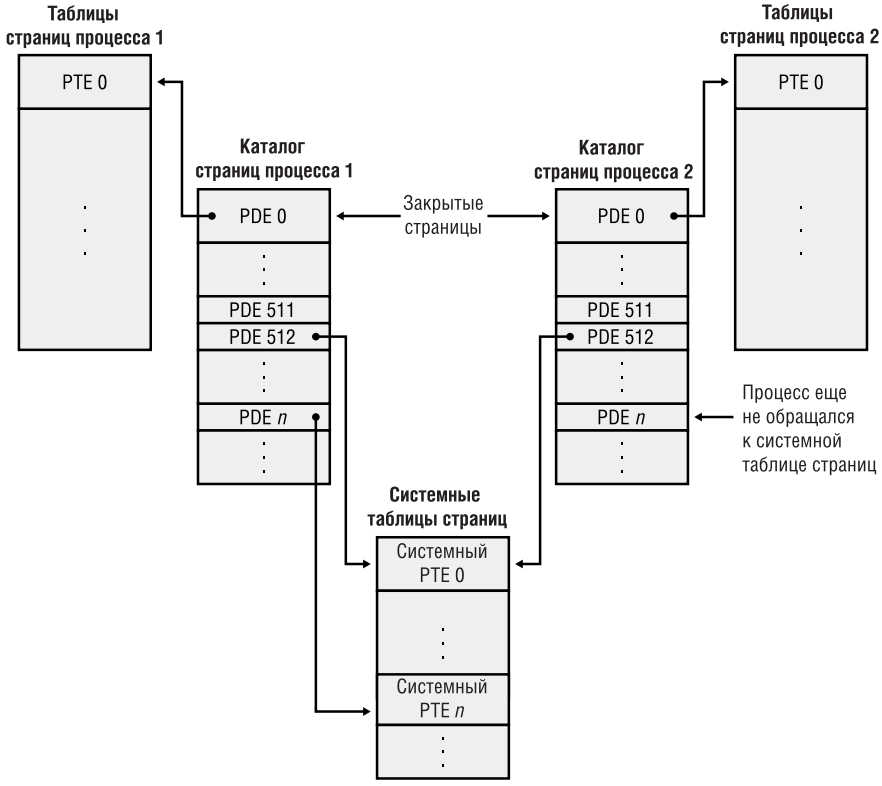


Рис. 7-18. *Системные таблицы страниц и таблицы страниц, принадлежащие процессам*

Таким образом, при обращении к пулу подкачиваемой памяти может возникнуть ошибка страницы из-за того, что каталог страниц процесса еще не содержит указатель на новую системную таблицу страниц, описывающую новую область пула. Но при доступе к пулу неподкачиваемой памяти таких ошибок не возникает, хотя он тоже может расширяться. Дело в том, что при инициализации системы Windows создает все системные таблицы страниц, которые описывают максимально возможный объем пула неподкачиваемой памяти.

Страницы таблиц и PTE

Элементы каталога страниц (page directory entries, PDE), принадлежащего процессу, указывают на индивидуальные таблицы страниц, которые состоят из массива PTE. Поле индекса таблицы страницы в виртуальном адресе (как показано на рис. 7-17) определяет PTE нужной страницы данных. В x86-системах размер этого индекса равен 10 битам (в PAE — 9), что позволяет ссылаться на 1024 4-байтных PTE (в PAE — на 512 8-байтных PTE). Но, поскольку 32-разрядная Windows предоставляет процессам 4-гигабайтное закрытое адресное пространство, для проецирования всего адресного пространства одной таблицы страниц мало. Чтобы подсчитать количество таблиц страниц, нужных для проецирования всех 4 Гб виртуального адресного пространства, поделите 4 Гб на объем виртуальной памяти, описываемой одной таблицей. Помните, что каждая таблица страниц в x86-системах определяет страницы данных суммарным размером в 4 Мб (в PAE — 2 Мб). Поэтому для проецирования всех 4 Гб адресного пространства требуется 1024 (4 Гб / 4 Мб) таблицы страниц, а в PAE-системах — 2048 (4 Гб / 2 Мб).

Для изучения PTE используйте команду *!pte* отладчика ядра (см. эксперимент «Трансляция адресов» далее в этой главе). Действительные PTE (здесь мы обсуждаем именно их — о недействительных PTE см. далее) состоят из двух основных полей (рис. 7-19): поля PFN физической страницы с данными (или физического адреса страницы в памяти) и поля флагов, описывающих состояние и атрибуты защиты страницы.

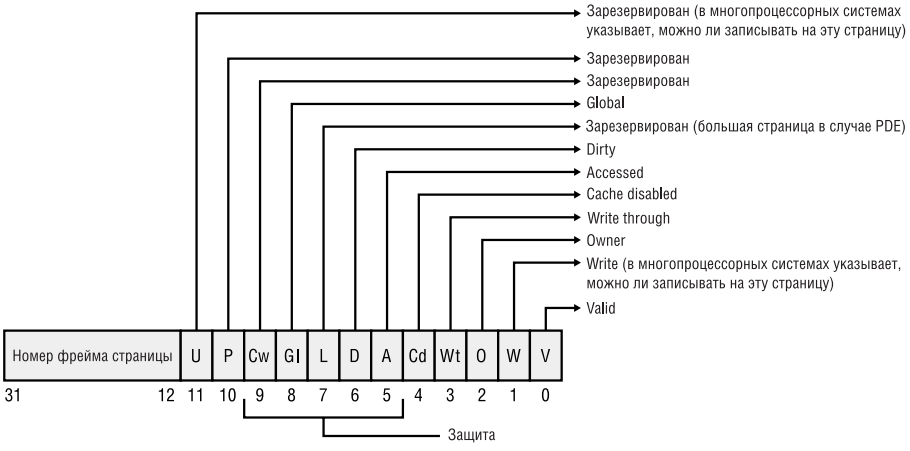


Рис. 7-19. *Действительные аппаратные PTE в x86-системах*

Как вы еще увидите, битовые флаги, помеченные как зарезервированные (рис. 7-19), используются, только если PTE недействителен (флаги интерпретируются программно). Аппаратно определяемые битовые флаги действительного PTE перечислены в таблице 7-11.

Таблица 7-11. *Битовые флаги PTE*

Битовые флаги	Описание
Accessed	Была операция чтения с данной страницы
Cache disabled	Кэширование данной страницы отключено
Dirty	Страница модифицирована
Global	Трансляция относится ко всем процессам (например, сброс буфера трансляции не повлияет на этот PTE)
Large page	Указывает, что PDE относится к 4-мегабайтной странице (или к 2-мегабайтной в PAE)
Owner	Указывает, доступна ли страница из кода пользовательского режима
Valid	Указывает, соответствует ли PTE странице в физической памяти
Write through	Отключает кэширование записи на данную страницу, в результате чего все измененные данные сразу же сбрасываются на диск
Write	В однопроцессорных системах указывает тип доступа (для чтения и записи или только для чтения), а в многопроцессорных системах определяет, доступна ли эта страница для записи (битовый флаг Write хранится в зарезервированной области PTE)

В x86-системах аппаратный PTE содержит биты Dirty и Accessed. Бит Accessed равен 0, если данные физической страницы, представляемой PTE, не были считаны или записаны. Процессор устанавливает этот бит при первой операции чтения или записи страницы. Бит Dirty устанавливается только после первой записи на страницу. Кроме того, бит Write обеспечивает защиту страницы: если он сброшен, страница доступна только для чтения, а если он установлен, страница доступна как для чтения, так и для записи. Когда поток пытается что-то записать на страницу со сброшенным битом Write, возникает исключение управления памятью, и обработчик, принадлежащий диспетчеру памяти, решает, может ли поток записывать данные на эту страницу (если она, например, помечена как копируемая при записи) или следует сгенерировать нарушение доступа.

Для аппаратных PTE в многопроцессорных x86-системах предусматривается дополнительный бит Write, реализуемый программно и предотвращающий остановку системы при сбросе кэша PTE (также называемого ассоциативным буфером трансляции). Этот бит указывает, что страница была модифицирована другим процессором.

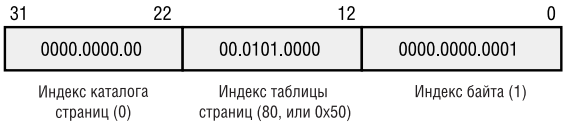
Адрес байта в пределах страницы

Как только диспетчер памяти находит искомую страницу, он переходит к поиску нужных данных на этой странице. На этом этапе используется поле индекса байта. Оно сообщает процессору, к какому байту данных на этой странице вы хотите обратиться. В x86-системах этот индекс состоит из 12 битов, что позволяет адресоваться максимум к 4096 байтам данных. Таким образом, добавление смещения байта к PFN, извлеченному из PTE, завершает трансляцию виртуального адреса в физический.

ЭКСПЕРИМЕНТ: трансляция адресов

Чтобы получше разобраться в том, как транслируются адреса, рассмотрим реальный пример трансляции виртуального адреса в x86-системе без поддержки PAE и с помощью отладчика ядра исследуем каталоги страниц, таблицы страниц и PTE. В этом примере мы используем процесс с виртуальным адресом 0x50001, спроецированным на действительный физический адрес. Как наблюдать за трансляцией недействительных адресов, мы поясним в последующих примерах.

Сначала преобразуем 0x50001 в двоичное значение и разобьем его на три поля, используемых при трансляции адреса. В двоичной системе счисления 0x50001 соответствует значению 101.0000.0000.0000.0001, а его поля выглядят так:



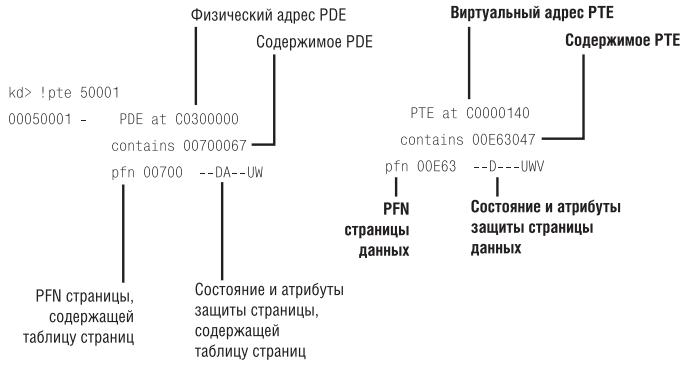
Чтобы начать трансляцию, процессор должен знать физический адрес каталога страниц, который хранится в регистре CR3, пока выполняется поток соответствующего процесса. Этот адрес можно получить как из регистра CR3, так и из дампа блока KPROCESS интересующего вас процесса с помощью команды *!process* отладчика ядра.

```
kd> !process
PROCESS 81555020 Cid: 0099 Peb: 7ffdf000  ParentCid: 0094
  DirBase: 012f0000  ObjectTable: 80695ba8  TableSize: 46.
  Image: IEXPLORE.EXE
        .
        .
        .
kd> r cr3
Cr3: 012F0000
```

Физический адрес каталога страниц

В данном случае физический адрес каталога страниц — 0x12F0000. Как видно на иллюстрации, поле индекса каталога страниц в этом примере равно 0. Поэтому физический адрес PDE — 0x12F0000.

Команда *!pte* отладчика ядра выводит PDE и PTE, описывающие виртуальный адрес:



В первой колонке отладчик ядра сообщает PDE, а во второй — PTE. Заметьте, что показывается виртуальный адрес PDE, а не физический. Как уже говорилось, каталог страниц процесса в x86-системах начинается с виртуального адреса 0xC0300000. Поскольку мы изучаем первый PDE каталога страниц, его адрес совпадает с адресом самого каталога.

Виртуальный адрес PTE равен 0xC0000140. Его можно вычислить, умножив индекс таблицы страниц (в данном случае — 0x50) на размер PTE (4), что дает 0x140. Поскольку диспетчер памяти проецирует таблицы страниц с адреса 0xC0000000, после добавления 140 получится виртуальный адрес, показанный на листинге: 0xC0000140. PFN страницы в каталоге страниц равен 0x700, а PFN страницы данных — 0xe63.

Флаги PTE показываются справа от PFN. Так, PTE, описывающий упомянутую выше страницу, имеет флаги D---UWV, где *D* обозначает dirty (данные страницы изменены), *U* — user-mode page (страница пользовательского режима), а *V* — valid (действительная страница).

Ассоциативный буфер трансляции

Как вы уже знаете, трансляция каждого адреса требует двух операций поиска: сначала нужно найти подходящую таблицу страниц в каталоге страниц, затем — элемент в этой таблице. Поскольку выполнение этих двух операций при каждом обращении по виртуальному адресу могло бы снизить быстродействие системы до неприемлемого уровня, большинство процессоров кэшируют транслируемые адреса, в результате чего необходимость в повторной трансляции при обращении к тем же адресам отпадает. Процессор поддерживает такой кэш в виде массива ассоциативной памяти, называемого *ассоциативным буфером трансляции* (translation look-aside buffer, TLB). Ассоциативная память вроде TLB представляет собой вектор, ячейки которого можно считывать и сразу сравнивать с целевым значением. В случае TLB вектор содержит сопоставления физических и виртуальных адресов для недавно использовавшихся страниц, а также атрибуты защиты каждой страницы, как показано на рис. 7-20. Каждый элемент TLB похож на элемент кэша, в метке которого хранятся компоненты виртуального адреса, а в поле

данных — номер физической страницы, атрибуты защиты, битовый флаг Valid и, как правило, битовый флаг Dirty. Эти флаги отражают состояние страницы, которой соответствует кэшированный PTE. Если в PTE установлен битовый флаг Global (используется для страниц системного пространства, глобально видимых всем процессам), то при переключениях контекста элемент TLB не объявляется недействительным.

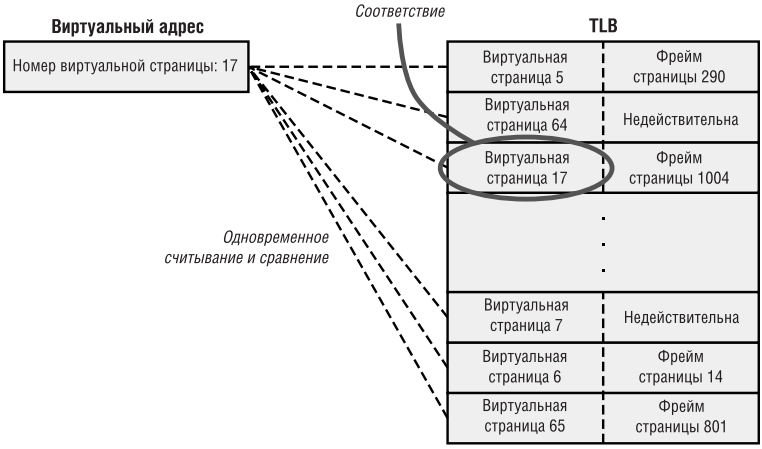


Рис. 7-20. *Применение ассоциативного буфера трансляции*

Часто используемым виртуальным адресам обычно соответствуют элементы в TLB, который обеспечивает чрезвычайно быструю трансляцию виртуальных адресов в физические, а в результате и быстрый доступ к памяти. Если виртуального адреса в TLB нет, он все еще может быть в памяти, но для его поиска понадобится несколько обращений к памяти, что увеличит время доступа. Если виртуальный адрес оказался в страничном файле или если диспетчер памяти изменил его PTE, диспетчер памяти должен явно объявить соответствующий элемент TLB недействительным. Если процесс повторно обращается к нему, генерируется ошибка страницы, нужная страница загружается обратно в память и для нее вновь создается элемент TLB.

Диспетчер памяти по возможности обрабатывает аппаратные и программные PTE одинаково. Так, при объявлении недействительного PTE действительным диспетчер памяти вызывает функцию ядра, которая обеспечивает аппаратно-независимую загрузку в TLB нового PTE. В x86-системах эта функция заменяется командой NOP, поскольку процессоры типа x86 самостоятельно загружают данные в TLB.

Physical Address Extension (PAE)

Режим проецирования памяти Physical Address Extension (PAE) впервые появился в x86-процессорах Intel Pentium Pro. При наличии соответствующей поддержки со стороны чипсета в режиме PAE можно адресоваться максимум к 64 Гб физической памяти на текущих x86-процессорах Intel и к 1024 Гб на

x64-процессорах (хотя в настоящее время Windows ограничивает этот показатель 128 Гб из-за размера базы данных PFN, которая понадобилась бы для проецирования такого большого объема памяти). При работе процессора в режиме PAE блок управления памятью (memory management unit, MMU) разделяет виртуальные адреса на 4 поля (рис. 7-21).

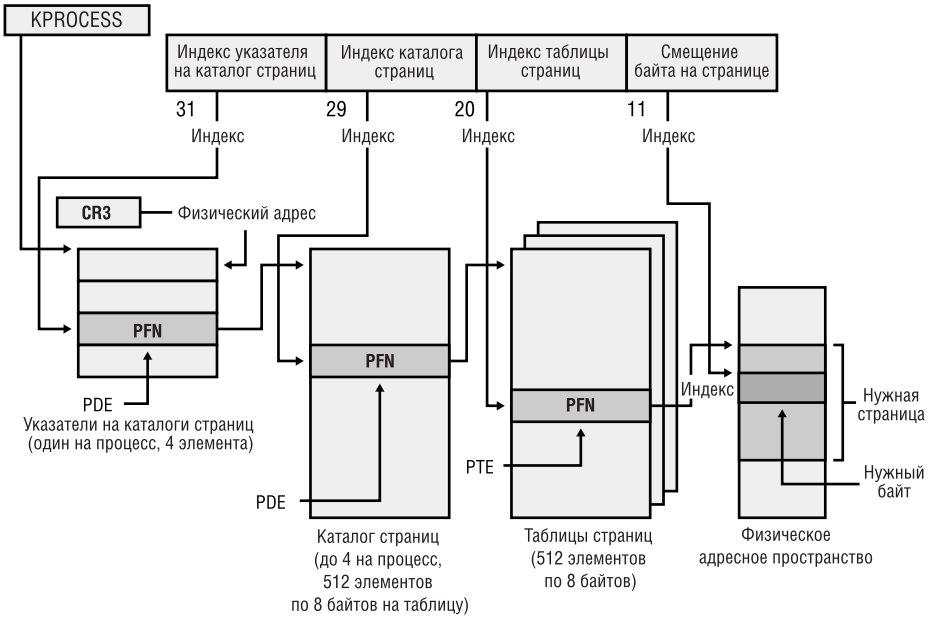


Рис. 7-21. *Проецирование страниц по механизму PAE*

При этом MMU по-прежнему реализует каталоги и таблицы страниц, но создает над ними третий уровень — таблицу указателей на каталоги страниц. PAE-режим позволяет адресовать больше памяти, чем стандартный, — но не из-за дополнительного уровня трансляции, а из-за большего размера PDE и PTE (по 64 бита вместо 32). Внутренне система представляет физический адрес 25 битами, что позволяет поддерживать максимум 2^{25+12} байтов, или 128 Гб, памяти. Для 32-разрядных приложений один из способов использования конфигураций с такими большими объемами памяти был представлен в разделе «Address Windowing Extensions» ранее в этой главе. Но, даже если приложения не обращаются к таким функциям, диспетчер памяти все равно задействует всю доступную физическую память под данные файлового кэша (см. раздел «База данных PFN» далее в этой главе).

Как мы поясняли в главе 2, существует специальная версия 32-разрядного ядра с поддержкой PAE — Ntkrnlpa.exe. Для загрузки этой версии ядра укажите в Boot.ini параметр /PAE. Заметьте, что она устанавливается во всех 32-разрядных системах Windows, даже в системах Windows 2000 Professional или Windows XP с малой памятью. Цель — упростить тестирование драйверов устройств. Поскольку в PAE-ядре драйверы устройств и другой системный код используют 64-разрядные адреса, загрузка с параметром /PAE по-

зволяет разработчикам тестировать свои драйверы на совместимость с системами, имеющими большие объемы памяти. Кстати, в связи с этим Boot.ini поддерживает еще один параметр — /NOLOWMEM, который запрещает использовать первые 4 Гб памяти (предполагается, что на компьютере установлено минимум 5 Гб физической памяти) и модифицирует адреса драйверов устройств для размещения выше этой границы, что гарантирует выход физических адресов драйверов за пределы 32-разрядных значений.

Трансляция виртуальных адресов на платформе IA64

Виртуальное адресное пространство на платформе IA64 аппаратно делится на восемь регионов. У каждого региона свой набор таблиц страниц. Windows использует только пять регионов, закрепляя таблицы страниц за тремя из них. Все регионы перечислены в таблице 7-12.

Таблица 7-12. *Регионы IA64*

Регион	Описание
0	Пользовательский код и данные
1	Код и данные пространства сеанса
2	Не используется
3	Не используется
4	Kseg3, который представляет собой кэшируемую проекцию «один к одному» (cached 1-to-1 mapping) физической памяти. Таблицы страниц для этого региона не нужны, так как операции вставки в TLB выполняются напрямую диспетчером памяти
5	Kseg4, который представляет собой некэшируемую проекцию «один к одному» (noncached 1-to-1 mapping) для физической памяти. Используется только в некоторых местах для доступа к адресам ввода-вывода, например к диапазонам портов ввода-вывода. Таблицы страниц для этого региона не нужны
6	Не используется
7	Код и данные ядра

При трансляции адресов 64-разрядной Windows на платформе IA64 используется трехуровневая схема таблиц страниц. Каждый процесс получает специальную структуру, содержащую 1024 указателя на каталоги страниц. Каждый каталог страниц содержит 1024 указателя на таблицы страниц, а те в свою очередь указывают на страницы физической памяти. Формат аппаратных PTE на платформе IA64 показан на рис. 7-22.

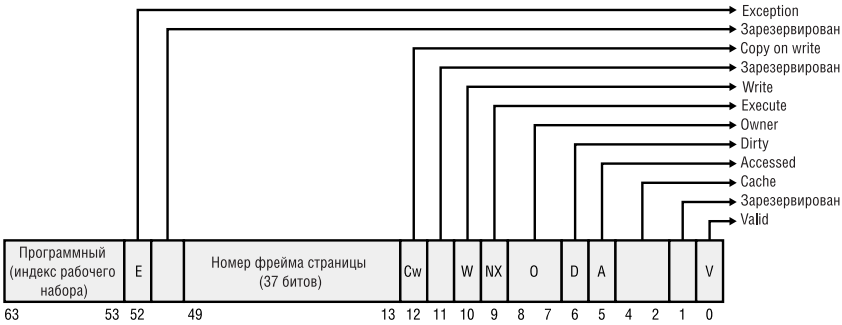


Рис. 7-22. *Аппаратные PTE в IA64-системах*

Трансляция виртуальных адресов на платформе x64

64-разрядная Windows на платформе x64 применяет четырехуровневую схему таблиц страниц. У каждого процесса имеется расширенный каталог страниц верхнего уровня (называемый картой страниц уровня 4), содержащий 512 указателей на структуру третьего уровня — родительский каталог страниц. Каждый родительский каталог страниц хранит 512 указателей на каталоги страниц второго уровня, а те содержат по 512 указателей на индивидуальные таблицы страниц. Наконец, таблицы страниц (в каждой из которых 512 PTE) указывают на страницы в памяти. В текущих реализациях архитектуры x64 размер виртуальных адресов ограничен 48 битами. Элементы 48-битного виртуального адреса представлены на рис. 7-23. Взаимосвязь между этими элементами показана на рис. 7-24, а формат аппаратного PTE на платформе x64 приведен на рис. 7-25.

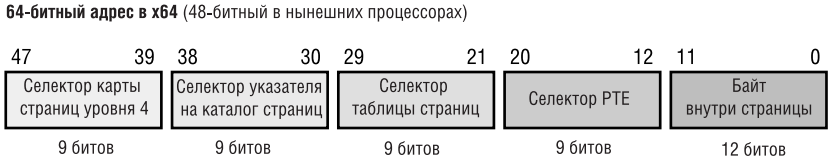


Рис. 7-23. *Виртуальный адрес на платформе x64*

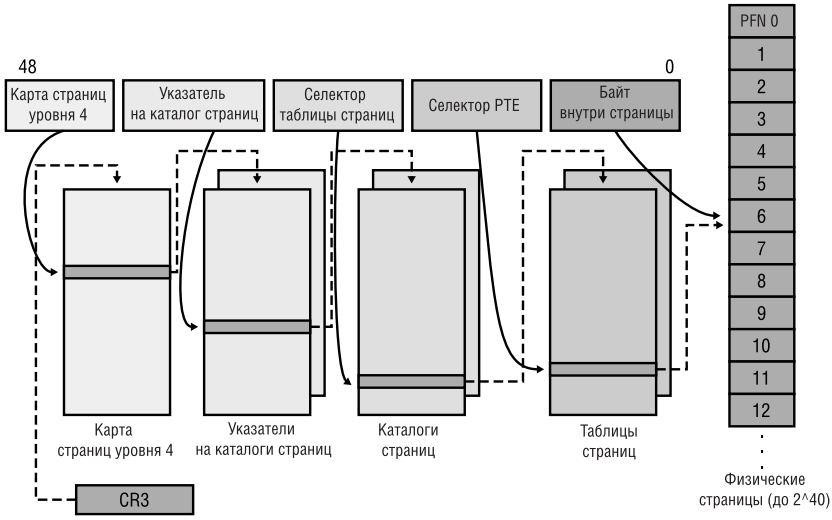


Рис. 7-24. *Структуры трансляции адресов в x64-системах*

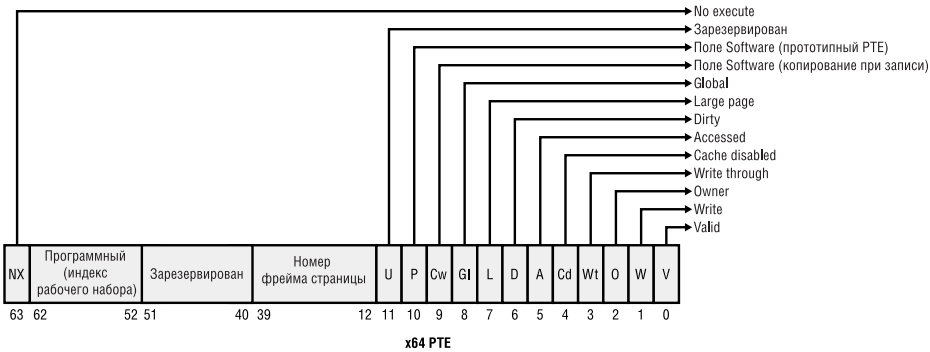


Рис. 7-25. *Аппаратные PTE в x64-системах*

Обработка ошибок страниц

Мы уже разобрались, как происходит трансляция адресов при действительных PTE. Если битовый флаг Valid в PTE сброшен, это значит, что нужная страница по какой-либо причине сейчас недоступна процессу. Здесь мы расскажем о типах недействительных PTE и о том, как разрешаются ссылки на такие PTE.

ПРИМЕЧАНИЕ В этой книге детально рассматриваются только PTE на 32-разрядной платформе x86. PTE для 64-разрядных систем содержат аналогичную информацию, но их подробную структуру мы не описываем.

При ссылке на недействительную страницу возникает *ошибка страницы* (page fault), и обработчик ловушки ядра (см. главу 3) перенаправляет ее

обработчику *MmAccessFault* диспетчера памяти. Последняя функция, выполняемая в контексте вызвавшего ошибку потока, предпринимает попытку ее разрешения (если это возможно) или генерирует соответствующее исключение. Причины таких ошибок перечислены в таблице 7-13.

Таблица 7-13. *Причины ошибок доступа к страницам*

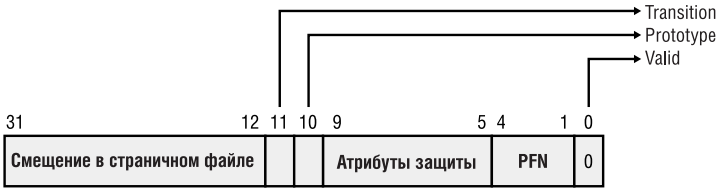
Причина	Результат
Обращение к странице, которая находится в страничном или проецируемом файле, но отсутствует в памяти	Выделение физической страницы и считывание нужной страницы с диска в рабочий набор
Обращение к странице из списка простаивающих (standby list) или модифицированных страниц (modified list)	Перемещение страницы в рабочий набор системы или процесса
Обращение к еще не переданной странице (например, к зарезервированному или еще не выделенному адресному пространству)	Нарушение доступа
Обращение из пользовательского режима к странице, доступной только из режима ядра	Нарушение доступа
Попытка записи на страницу, доступную только для чтения	Нарушение доступа
Обращение к странице, обнуляемой по требованию (demand-zero page)	Добавление страницы, заполненной нулями, в рабочий набор процесса
Попытка записи на сторожевую страницу (guard page)	Нарушение доступа (если ссылка относилась к стеку пользовательского режима, он автоматически расширяется)
Попытка записи на страницу с атрибутом «копирование при записи»	Создание копии этой страницы для процесса (или сеанса) и замена ею исходной страницы в рабочем наборе процесса, сеанса или системы
Ссылка на действительную страницу в системном пространстве, но отсутствующую в каталоге страниц процесса (например при расширении подкачиваемого пула уже после того, как был создан каталог страниц процесса)	Копирование PDE из главного системного каталога страниц и закрытие исключения
Запись на действительную, но пока не изменявшуюся страницу в многопроцессорной системе	Установка битового флага Dirty в PTE
Попытка выполнения кода на странице с атрибутом защиты «запрет на выполнение»	Нарушение доступа [только на аппаратных платформах с поддержкой защиты «запрет на выполнение» (no execute protection) под управлением Windows XP Service Pack 2 или Windows Server 2003 Service Pack 1 и выше]

В следующем разделе описываются четыре базовых типа недействительных PTE. Затем мы рассмотрим особый случай недействительных PTE — прототипные PTE, используемые для поддержки разделяемых страниц.

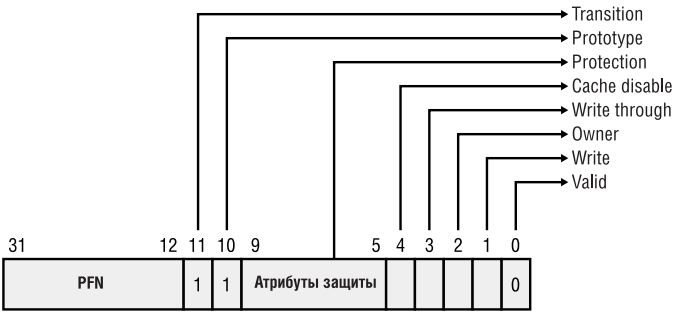
Недействительные PTE

Ниже приведен список типов недействительных PTE с описанием их структуры. Некоторые их флаги идентичны флагам аппаратных PTE (см. таблицу 7-11).

- **PTE для страницы в страничном файле (page file PTE)** Нужная страница находится в страничном файле. Инициируется операция загрузки страницы.



- **PTE для страницы, обнуляемой по требованию (demand zero PTE)** Нужная страница должна быть заполнена нулями. Сначала просматривается список обнуленных страниц (zero page list). Если он пуст, просматривается список свободных страниц (free list). Если в нем есть свободная страница, она заполняется нулями. Если этот список тоже пуст, используется список простаивающих страниц (stanby list). Формат этого PTE идентичен формату PTE для страницы в страничном файле, но номер страничного файла и смещение в нем равны 0.
- **Переходный PTE (transition PTE)** Нужная страница находится в памяти в списке простаивающих, модифицированных (modified list) или модифицированных, но не записываемых страниц (modified-no-write list). Страница будет удалена из списка и добавлена в рабочий набор, как только на нее будет ссылка.



- **Неизвестный PTE (unknown PTE)** PTE равен 0, либо таблицы страниц еще нет. В обоих случаях этот флаг означает, что определить, передана ли память по данному адресу, можно только через дескрипторы виртуальных адресов (VAD). Если передана, то формируются таблицы страниц, представляющие новую область адресного пространства, которому

передана физическая память. (Описание VAD см. в разделе «Дескрипторы виртуальных адресов» далее в этой главе.)

Прототипные PTE

Если какая-то страница может разделяться двумя процессами, то при проецировании таких потенциально разделяемых страниц диспетчер памяти использует структуру, называемую прототипным PTE (prototype page table entry). В случае разделов, поддерживаемых страничными файлами (page file backed sections), массив прототипных PTE формируется при первом создании объекта «раздел», а в случае проецируемых файлов этот массив создается порциями при проецировании каждого представления. Прототипные PTE являются частью структуры сегмента, описываемой в конце этой главы.

ПРИМЕЧАНИЕ В Windows 2000 и Windows 2000 Service Pack 1 диспетчер памяти создает все прототипные PTE, нужные для проецирования всего файла, даже если приложение единовременно проецирует представления лишь на небольшие части файла. Поскольку эти структуры создаются в конечном ресурсе (в пуле подкачиваемой памяти), попытка спроецировать большие файлы может привести к истощению этого ресурса. В итоге предельный общий объем единовременно используемых проецируемых файлов составляет около 200 Гб.

Этот лимит снят в Windows 2000 Service Pack 2 и более поздних версиях за счет того, что диспетчер памяти теперь создает такие структуры только при создании проецируемых на файл представлений. Благодаря этому стало возможным резервное копирование огромных файлов даже на компьютерах с малым объемом памяти.

Когда процесс впервые ссылается на страницу, проецируемую на представление объекта «раздел» (вспомните, что VAD создаются только при проецировании представления), диспетчер памяти — на основе информации из прототипного PTE — заполняет реальный PTE, используемый для трансляции адресов в таблице страниц процесса. Когда разделяемая страница становится действительной, PTE процесса и прототипный PTE указывают на физическую страницу с данными. Для учета числа PTE, ссылающихся на действительные разделяемые страницы, в базе данных PFN увеличивается значение соответствующего счетчика (см. раздел «База данных PFN» далее в этой главе). Благодаря этому диспетчер памяти сможет определить тот момент, когда на разделяемую страницу больше не будет ссылок ни в одной таблице страниц, а затем объявить ее недействительной и поместить в список переходных страниц или выгрузить на диск.

Как только разделяемая страница объявлена недействительной, PTE в таблице страниц процесса заменяется особым PTE, указывающим на прототипный PTE, который описывает данную страницу (рис. 7-26).

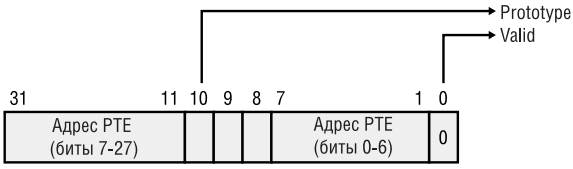


Рис. 7-26. *Структура недействительного PTE, указывающего на прототипный PTE*

Таким образом, при последующем обращении к странице диспетчер памяти, используя информацию из особого PTE, может найти прототипный PTE, который в свою очередь описывает нужную страницу. Разделяемая страница может находиться в одном из шести состояний, указанных в прототипном PTE.

- **Активная/действительная (active/valid)** Страница находится в физической памяти в результате обращения к ней другого процесса.
- **Переходная (transition)** Страница находится в памяти в списке простаивающих или модифицированных страниц.
- **Модифицированная, но не записываемая (modified-no-write)** Страница находится в памяти в списке модифицированных, но не записываемых страниц (см. таблицу 7-20).
- **Обнуляемая по требованию (demand zero)** Страницу требуется обнулить (заполнить нулями).
- **Выгруженная в страничный файл (page file)** Страница находится в страничном файле.
- **Содержащаяся в проецируемом файле (mapped file)** Страница находится в проецируемом файле.

Хотя формат прототипных PTE идентичен формату реальных PTE, они используются не для трансляции адресов, а как уровень между таблицей страниц и базой данных PFN и никогда не записываются непосредственно в таблицы страниц.

Заставляя всех пользователей потенциально разделяемой страницы ссылаться на прототипный PTE, диспетчер памяти может управлять разделяемыми страницами, не обновляя таблицы страниц в каждом процессе. Допустим, в какой-то момент разделяемая страница выгружается в страничный файл на диске. При ее загрузке обратно в память диспетчеру памяти понадобится изменить только прототипный PTE, записав в него указатель на новый физический адрес страницы, а PTE в таблицах страниц всех процессов, совместно использующих эту страницу, останутся прежними (в этих PTE битовый флаг Valid сброшен, они ссылаются на прототипный PTE). Реальные PTE обновляются позднее, по мере обращения процессов к этой странице.

На рис. 7-27 показаны две виртуальные страницы в проецируемом представлении. Одна из них действительна, другая — нет. Как видите, на действительную страницу ссылаются PTE процесса и прототипный PTE. Недействительная страница находится в страничном файле, ее точный адрес определяется прототипным PTE. PTE данного процесса (как и любого другого процесса, проецирующего эту страницу) содержит указатель на прототипный PTE.

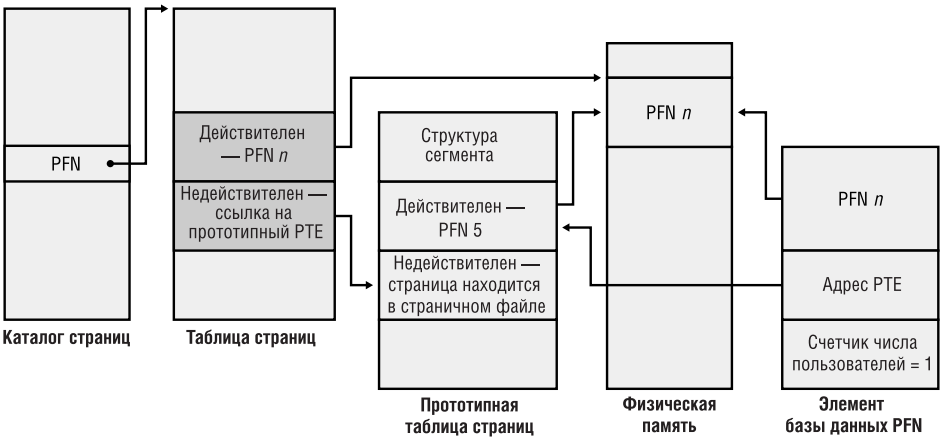


Рис. 7-27. *Прототипные PTE*

Операции ввода-вывода, связанные с подкачкой страниц

Такие операции ввода-вывода происходят в результате запроса на чтение страничного или проецируемого файла из-за ошибки страницы. Кроме того, поскольку в страничный файл могут помещаться и таблицы страниц, обработка ошибки страницы в случае таблицы страниц может повлечь за собой новые ошибки страниц.

Операции ввода-вывода, связанные с подкачкой, являются синхронными, т. е. поток ждет завершения подобной операции на каком-либо событии и она не может быть прервана вызовом асинхронной процедуры (APC). Для идентификации ввода-вывода как связанного с подкачкой подсистема подкачки страниц (pager) вызывает функцию запроса ввода-вывода, указывая специальный модификатор. По завершении операции подсистема ввода-вывода освобождает событие. Это пробуждает подсистему подкачки страниц, и она продолжает свою работу.

В ходе операции ввода-вывода, связанной с подкачкой, поток, который вызвал ошибку страницы, не владеет критичными синхронизирующими объектами, используемыми при управлении памятью. Другие потоки того же процесса могут вызывать функции управления виртуальной памятью и обрабатывать ошибки страниц в ходе операции ввода-вывода, связанной с подкачкой. Однако подсистема подкачки страниц должна уметь выходить из некоторых ситуаций, которые могут возникать на момент завершения такой операции:

- другой поток в том же или другом процессе вызывает ошибку той же страницы, из-за чего происходит конфликт ошибок страницы (см. следующий раздел);
- страница удалена из виртуального адресного пространства и перепроецирована;

- сменился атрибут защиты страницы;
- ошибка относится к прототипному PTE, а страница, которая проецирует этот PTE, отсутствует в рабочем наборе.

Подсистема подкачки страниц выходит из таких ситуаций следующим образом. Перед запросом на операцию ввода-вывода, связанную с подкачкой, она сохраняет в стеке ядра потока статусную информацию, что позволяет после выполнения запроса распознать возникновение одной из перечисленных выше ситуаций и при необходимости отбросить ошибку страницы, не делая эту страницу действительной. Если команда, вызвавшая ошибку страницы, выдается повторно, вновь активизируется подсистема подкачки страниц, и PTE вычисляется заново.

Конфликты ошибок страницы

Конфликт ошибок страницы (collided page fault) возникает, когда другой поток или процесс вызывает ошибку страницы, уже обрабатываемой в данный момент из-за предыдущей ошибки того же типа. Подсистема подкачки страниц распознает и оптимальным образом разрешает такие конфликты, поскольку они нередки в системах с поддержкой многопоточности. Если другой поток или процесс вызывает ошибку той же страницы, подсистема подкачки страниц обнаруживает конфликт ошибок страницы, отмечая при этом, что страница находится в переходном состоянии и что она сейчас считывается. (Эта информация извлекается из элемента базы данных PFN.) Далее подсистема подкачки страниц переходит в ожидание на событии, указанном в элементе базы данных PFN. Это событие было инициализировано потоком, вызвавшим первую ошибку страницы.

По завершении операции ввода-вывода событие переходит в свободное состояние. Первый поток, захвативший блокировку базы данных PFN, отвечает за заключительные операции, связанные с подкачкой. К ним относятся проверка статуса операции ввода-вывода (чтобы убедиться в ее успешном завершении), сброс бита «в процессе чтения» в базе данных PFN и обновление PTE.

Когда следующие потоки захватывают блокировку базы данных PFN для завершения обработки конфликтующих ошибок страницы, сброшенный бит «в процессе чтения» сообщает подсистеме подкачки страниц, что начальное обновление закончено, и она проверяет флаг ошибок в элементе базы данных PFN. Если этот флаг установлен, PTE не обновляется, и в потоке, вызвавшем ошибку страницы, генерируется исключение «in-page error» (ошибка в процессе загрузки страницы).

Страничные файлы

Страничные файлы (page files) предназначены для хранения модифицированных страниц, которые используются каким-то процессом, но должны быть выгружены из памяти на диск. Пространство в страничном файле резервируется, когда происходит начальная передача страниц, но реальные участки страничного файла не выбираются до тех пор, пока страницы не

выгружаются на диск. Важно отметить, что система накладывает ограничение на число передаваемых закрытых страниц. Поэтому значение счетчика производительности Process: Page File Bytes на самом деле отражает суммарный объем закрытой памяти, переданной процессам. Соответствующие страницы могут находиться в страничном файле (частично или целиком) или, напротив, в физической памяти. (В сущности этот счетчик идентичен счетчику Process: Private Bytes.)

Диспетчер памяти отслеживает использование закрытой переданной памяти на глобальном уровне и по каждому процессу отдельно (в виде квоты страничного файла). И вновь эти данные отражают не размер использованного пространства в страничном файле, а объем переданной закрытой памяти. Соответствующие счетчики увеличиваются при передаче виртуальных адресов, требующих новых закрытых физических страниц. Как только система достигнет глобального лимита на переданную память (т. е. физическая память и страничные файлы заполнены), попытки выделения виртуальной памяти будут заканчиваться неудачно — пока какой-либо процесс не освободит переданную ему память (например, после завершения).

При загрузке системы процесс диспетчера сеансов (см. главу 4) считывает список страничных файлов, которые он должен открыть. Этот список хранится в параметре реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagingFiles. Этот многострочный параметр содержит имя, минимальный и максимальный размеры каждого страничного файла. Windows поддерживает до 16 страничных файлов. В x86-системах с обычным ядром каждый страничный файл может быть размером до 4095 Мб, в x64- и x86-системах с PAE-ядром — до 16 Тб, а в IA64-системах — до 32 Тб. Страничные файлы нельзя удалить во время работы системы, так как процесс System (см. главу 2) открывает описатель каждого страничного файла. Тот факт, что страничные файлы открываются системой, объясняет, почему встроенное средство дефрагментации не в состоянии дефрагментировать страничный файл в процессе работы системы. Для дефрагментации страничного файла используйте бесплатную утилиту Pagedefrag с сайта *www.sysinternals.com*. В ней применяется тот же подход, что и в других сторонних утилитах дефрагментации: она запускает свой процесс дефрагментации на самом раннем этапе загрузки системы, еще до открытия страничных файлов диспетчером сеансов.

Поскольку страничный файл содержит части виртуальной памяти процессов и ядра, для большей безопасности его можно настроить на очистку при выключении системы. Для этого установите параметр реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\ClearPageFileAtShutdown в 1. Иначе в страничном файле останутся те данные, которые были выгружены в него к моменту выключения системы. И к этим данным сможет обратиться любой, кто получит физический доступ к компьютеру.

Если не указано ни одного страничного файла, Windows 2000 создает в загрузочном разделе 20-мегабайтный страничный файл. Windows XP и Windows Server 2003 не создают этот временный страничный файл, и поэтому

в такой ситуации объем системной виртуальной памяти будет ограничен доступной физической памятью. Windows XP и Windows Server 2003, если минимальный и максимальный размеры страничного файла заданы нулевыми, считают, что этот файл управляется системой, и его размер выбирается в соответствии с данными, показанными в таблице 7-14.

Таблица 7-14. *Размеры страничного файла по умолчанию*

Объем оперативной памяти (RAM)	Минимальный размер страничного файла	Максимальный размер страничного файла
Менее 1 Гб	1,5 * RAM	3 * RAM
1 Гб и более	1 * RAM	3 * RAM

ЭКСПЕРИМЕНТ: просмотр страничных файлов

Как уже говорилось, список страничных файлов хранится в параметре реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagingFiles. Он содержит конфигурационные параметры страничных файлов, которые модифицируются через аплет System (Система) в Control Panel (Панель управления). В Windows 2000 щелкните кнопку Performance Options (Параметры быстродействия) на вкладке Advanced (Дополнительно) и нажмите кнопку Change (Изменить). В Windows XP и Windows Server 2003 откройте вкладку Advanced (Дополнительно), щелкните кнопку Settings (Параметры) в разделе Performance (Быстродействие), откройте еще одну вкладку Advanced (Дополнительно) и, наконец, нажмите кнопку Change (Изменить) в разделе Virtual Memory (Виртуальная память).

Создать новый страничный файл можно через Control Panel. При этом вызывается системный сервис *NtCreatePagingFile*, определенный в Ntdll.dll и предназначенный только для внутреннего использования. Страничные файлы всегда создаются несжатыми, даже если находятся в сжатом каталоге. Для защиты новых страничных файлов от удаления их описатели дублируются в процесс System.

В таблице 7-15 перечислены счетчики производительности, с помощью которых можно исследовать использование переданной закрытой памяти в рамках как всей системы, так и каждого страничного файла. К сожалению, определить соотношение резидентной и нерезидентной (находящейся в страничном файле) частей закрытой памяти, которая передана какому-либо процессу, нельзя.

Заметьте, что эти счетчики могут помочь в подборе размера страничного файла. Исходить из объема оперативной памяти (RAM) нет смысла: чем больше у вас памяти, тем меньше вероятность того, что вам понадобится выгрузка данных на диск. Чтобы определить, какой размер страничного файла действительно нужен в вашей системе с учетом используемых вами приложений, проверьте пиковое значение переданной памяти, которое отображается в разделе Commit Charge (Выделение памяти) на вкладке Performance

(Быстродействие) диспетчера задач, а также в окне System Information утилиты Process Explorer. Этот показатель отражает пиковый объем страничного файла с момента загрузки системы, который понадобился бы в том случае, если бы системе пришлось выгрузить всю закрытую переданную виртуальную память (что происходит крайне редко).

Таблица 7-15. *Счетчики, относящиеся к переданной памяти и страничному файлу*

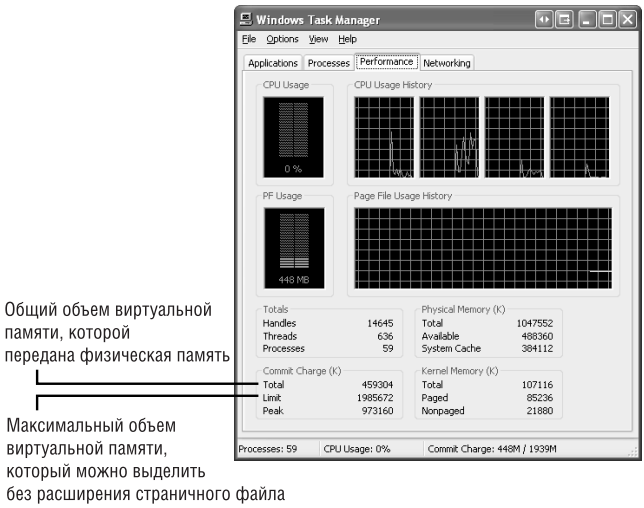
Счетчик	Описание
Memory: Committed Bytes (Память: Байт выделенной виртуальной памяти)	Число байтов переданной виртуальной (не зарезервированной) памяти. Это значение не обязательно отражает использование страничного файла, поскольку включает закрытые переданные страницы в физической памяти, никогда не выгружавшиеся в страничный файл. Так что оно скорее указывает, сколько места в страничном файле понадобилось бы в том случае, если бы процесс был полностью выгружен из памяти
Memory: Commit Limit (Память: Предел выделенной виртуальной памяти)	Число байтов виртуальной памяти, которое может быть передано без расширения страничных файлов. Если страничные файлы можно расширять, этот лимит не является жестким
Paging File: % Usage (Файл подкачки: % использования)	Процентная доля памяти, переданной из страничного файла
Paging File: % Usage Peak [Файл подкачки: % использования (пик)]	Пиковое значение процентной доли памяти, переданной из страничного файла

Если страничный файл в вашей системе слишком велик, Windows не будет использовать лишнее пространство; иначе говоря, увеличение размера страничного файла не изменит производительность системы — просто у нее будет больше неразделяемой (non-shareable) переданной виртуальной памяти. Но если страничный файл слишком мал для запускаемого вами набора приложений, может появиться сообщение об ошибке «system running low on virtual memory» (в системе не хватает виртуальной памяти). В таком случае сначала проверьте, не дает ли какой-нибудь процесс утечки памяти. Для этого посмотрите на счетчики байтов закрытой памяти для процессов в столбце VM Size (Объем виртуальной памяти) на вкладке Processes (Процессы) диспетчера задач. Если ни один из процессов вроде бы не дает утечки памяти, проделайте операции, описанные в эксперименте «Анализ утечки памяти в пуле» ранее в этой главе.

ЭКСПЕРИМЕНТ: наблюдаем за использованием страничного файла через диспетчер задач

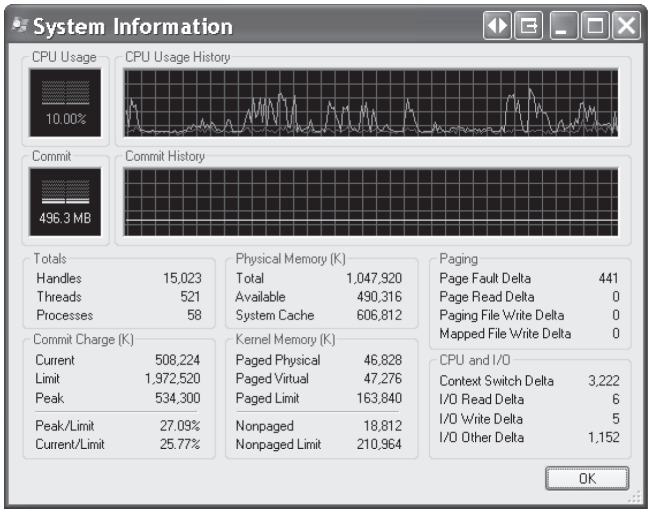
Вы можете узнать, как используется переданная память, и с помощью Task Manager (Диспетчер задач), открыв в нем вкладку Performance (Быстродействие). При этом вы увидите следующие счетчики, связанные со страничными файлами.

см. след. стр.



Заметьте, что график Mem Usage, который в Windows XP и Windows Server 2003 называется PF Usage (Файл подкачки), на самом деле соответствует общему объему переданной системной памяти (system commit total). Это значение отражает *потенциально возможное*, а не реальное использование страничного файла. Как мы уже говорили, столько места в страничном файле понадобилось бы в том случае, если бы системе вдруг пришлось выгрузить сразу всю закрытую переданную виртуальную память.

Дополнительную информацию вы найдете в окне System Information утилиты Process Explorer.



Дескрипторы виртуальных адресов

Момент загрузки страниц в память диспетчер памяти определяет, используя алгоритм подкачки по требованию (demand-paging algorithm). Страница загружается с диска, если поток, обращаясь к ней, вызывает ошибку страницы. Подобно копированию при записи подкачка по требованию является одной из форм *отложенной оценки* (lazy evaluation) — выполнения операции только при ее абсолютной необходимости.

Диспетчер памяти использует отложенную оценку не только при загрузке страниц в память, но и при формировании таблиц, описывающих новые страницы. Например, когда поток передает память большой области виртуальной памяти с помощью *VirtualAlloc*, диспетчер памяти мог бы немедленно создать таблицы страниц, необходимые для доступа ко всему диапазону выделенной памяти. А что если часть этого диапазона останется невостребованной? Зачем впустую расходовать процессорное время? Вместо этого диспетчер памяти откладывает формирование таблицы страниц до тех пор, пока поток не вызовет ошибку страницы. Такой подход существенно увеличивает быстродействие процессов, резервирующих и/или передающих большие объемы памяти, но обращающихся к ней не очень часто.

При использовании алгоритма отложенной оценки выделение даже больших блоков памяти происходит очень быстро. Когда поток выделяет память, диспетчер памяти должен соответственно отреагировать. Для этого диспетчер памяти поддерживает набор структур данных, которые позволяют вести учет зарезервированных и свободных виртуальных адресов в адресном пространстве процесса. Эти структуры данных называются *дескрипторами виртуальных адресов* (virtual address descriptors, VAD). Для каждого процесса диспетчер памяти поддерживает свой набор VAD, описывающий состояние адресного пространства этого процесса. Для большей эффективности поиска VAD организованы в виде двоичного дерева с автоматической балансировкой. В Windows Server 2003 реализован алгоритм дерева AVL (это первые буквы фамилий его разработчиков — Adelson-Velskii и Landis), который обеспечивает более эффективную балансировку VAD-дерева, а это уменьшает среднее число операций сравнения при поиске VAD, соответствующего некоему виртуальному адресу. Схема дерева VAD показана на рис. 7-28.

Когда процесс резервирует адресное пространство или проецирует представление раздела, диспетчер памяти создает VAD для хранения информации из запроса на выделение — диапазона резервируемых адресов, его типа (разделяемый или закрытый), возможности наследования содержимого диапазона дочерними процессами, атрибутов защиты, установленных для страниц этого диапазона.

При первом обращении потока по какому-либо адресу диспетчер памяти должен создать PTE страницы, содержащей данный адрес. Для этого он находит VAD, чей диапазон включает нужный адрес, и использует его информацию для заполнения PTE. Если адрес выпадает из диапазонов VAD или находится в зарезервированном, но не переданном диапазоне адресов, диспет-

чер памяти узнает, что поток не выделил память до попытки ее использования, и генерирует нарушение доступа.

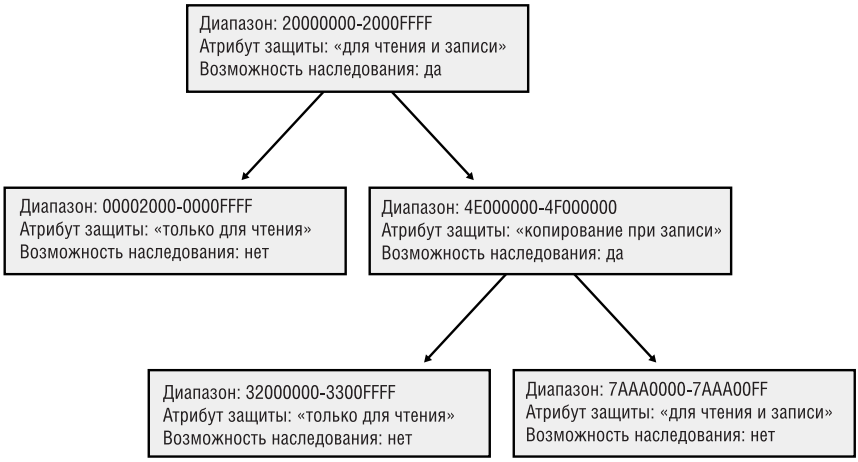


Рис. 7-28. *Дескрипторы виртуальных адресов*

ЭКСПЕРИМЕНТ: просмотр дескрипторов виртуальных адресов

Чтобы просмотреть VAD для какого-либо процесса, используйте команду *!vad* отладчика ядра. Сначала найдите адрес корня VAD-дерева с помощью команды *!process*. Затем введите полученный адрес в команде *!vad*, как показано в примере для процесса, выполняющего Notepad.exe.

```
kd> !process 2a0 1
Searching for Process with Cid == 2a0
PROCESS 8614d030 SessionId: 0 Cid: 02a0 Peb: 7ffdf000 ParentCid: 0554
    DirBase: 00d93000  ObjectTable: 81bc47c8  TableSize:  41.
    Image: notepad.exe
    VadRoot 8118d868 Clone 0 Private 252. Modified 0. Locked 0.
    ...

kd> !vad 8118d868
VAD     level      start      end    commit
84df4148 ( 2)         10       10         1 Private      READWRITE
850cdbe8 ( 3)         20       20         1 Private      READWRITE
810b0ee8 ( 1)         30       6f         7 Private      READWRITE
8109d308 ( 3)         70      16f        32 Private      READWRITE
810e9a28 ( 2)        170      17f         0 Mapped       READWRITE
84aedfc8 ( 3)        180      195         0 Mapped       READONLY
8118d868 ( 0)        1a0      1ce         0 Mapped       READONLY
81190a08 ( 4)        1d0      210         0 Mapped       READONLY
85c7b928 ( 3)        220      223         0 Mapped       READONLY
86253a08 ( 4)        230      2f7         0 Mapped       EXECUTE_READ
810aab48 ( 2)        300      342         0 Mapped       READONLY
```

```
80db5448 ( 5)        350      64f         0 Mapped       EXECUTE_READ
...

Total VADs:    49  average level:    6  maximum depth: 13
```

Объекты-разделы

Вероятно, вы помните, что объект «раздел» (section object), в подсистеме Windows называемый объектом «проекция файла» (file mapping object), представляет блок памяти, доступный двум и более процессам для совместного использования. Объект-раздел можно проецировать на страничный файл или другой файл на диске.

Исполнительная система использует разделы для загрузки исполняемых образов в память, а диспетчер кэша — для доступа к данным в кэшированном файле (подробнее на эту тему см. главу 11). Объекты «раздел» также позволяют проецировать файлы на адресные пространства процессов. При этом можно обращаться к файлу как к большому массиву, проецируя разные представления объекта-раздела и выполняя операции чтения-записи в памяти, а не в самом файле, — такие операции называются *вводом-выводом в проецируемые файлы* (mapped file I/O). Если программа обратится к недействительной странице (отсутствующей в физической памяти), возникнет ошибка страницы, и диспетчер памяти автоматически загрузит эту страницу в память из проецируемого файла. Если программа модифицирует страницу, диспетчер памяти сохранит изменения в файле в процессе обычных операций, связанных с подкачкой. (Приложение может самостоятельно сбросить представление файла на диск вызовом Windows-функции *FlushViewOfFile*.)

Как и другие объекты, разделы создаются и уничтожаются диспетчером объектов. Он создает и инициализирует заголовок объекта «раздел», а диспетчер памяти определяет тело этого объекта. Диспетчер памяти также реализует сервисы, через которые потоки пользовательского режима могут получать и изменять атрибуты, хранящиеся в теле объекта «раздел». Структура объекта «раздел» показана на рис. 7-29.

Рис. 7-29. *Объект «раздел»*

Уникальные атрибуты, хранящиеся в объектах «раздел» перечислены в таблице 7-16.

Таблица 7-16. *Атрибуты, хранящиеся в теле объекта-раздела*

Атрибут	Описание
Максимальный размер	Предельный размер, до которого может увеличиваться раздел (в байтах). При проецировании файла максимальный размер не может быть больше длины файла
Атрибуты защиты страниц	Атрибуты защиты, назначенные всем страницам раздела при его создании
Страничный/проецируемый файл	Указывает, создан ли раздел пустым (как поддерживаемый страничным файлом) или в него загружен файл (как поддерживаемый проецируемым файлом)
Базовый/не базовый	Указывает, является ли раздел базовым (загружаемым по одному и тому же виртуальному адресу для всех использующих его процессов) или нет (при необходимости загружаемым по разным виртуальным адресам для разных процессов)

ЭКСПЕРИМЕНТ: просмотр объектов «раздел»

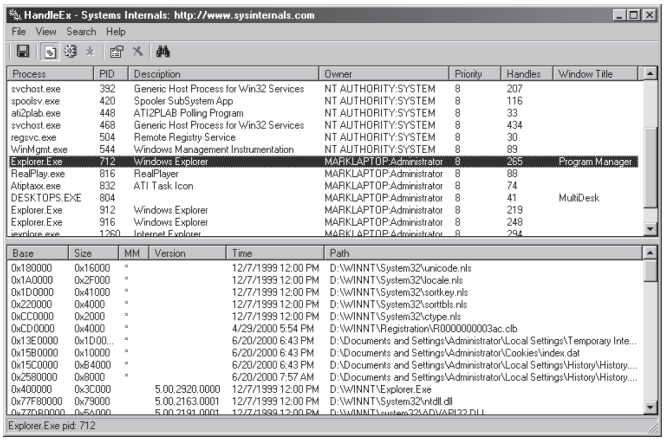
Утилита Object Viewer (Winobj.exe с сайта *www.sysinternals.com* или Winobj.exe из Platform SDK) позволяет просмотреть список разделов с глобальными именами. Вы можете перечислить открытые описатели объектов «раздел» с помощью любых утилит, описанных в разделе «Диспетчер объектов» главы 3 и способных перечислять содержимое таблицы открытых описателей. (Как уже говорилось в главе 3, эти имена хранятся в каталоге диспетчера объектов \BaseNamedObjects.)

Используя Process Explorer или Handles.exe (*www.sysinternals.com*), либо утилиту Oh.exe (Open Handles) из ресурсов Windows, можно вывести список открытых описателей объектов «раздел». Например, следующая команда показывает все открытые описатели каждого объекта «раздел» независимо от того, есть ли у него имя. (Разделу должно быть присвоено имя, если другой процесс открывает его по имени.)

```
c:\> oh -t section -a
00000008 System        Section  0070
0000008C smss.exe      Section  0004
000000A4 csrss.exe     Section  0004
000000A4 csrss.exe     Section  0024
000000A4 csrss.exe     Section  0038
000000A4 csrss.exe     Section  0040 \NLS\NlsSectionUnicode
000000A4 csrss.exe     Section  0044 \NLS\NlsSectionLocale
000000A4 csrss.exe     Section  0048 \NLS\NlsSectionCType
000000A4 csrss.exe     Section  004c \NLS\NlsSectionSortkey
000000A4 csrss.exe     Section  0050 \NLS\NlsSectionSortTbls
000000A0 winlogon.exe  Section  0004
000000A0 winlogon.exe  Section  0034
```

```
000000A0 winlogon.exe  Section  0168 \BaseNamedObjects\mmGlobalPnpInfo
...
```

Для просмотра проецируемых файлов можно воспользоваться и утилитой Process Explorer. Выберите из меню View команду Lower Pane View, а затем DLLs. Файлы в колонке MM, помеченные звездочкой, являются проецируемыми (в отличие от DLL и других файлов, загружаемых загрузчиком образов в виде модулей). Вот пример:



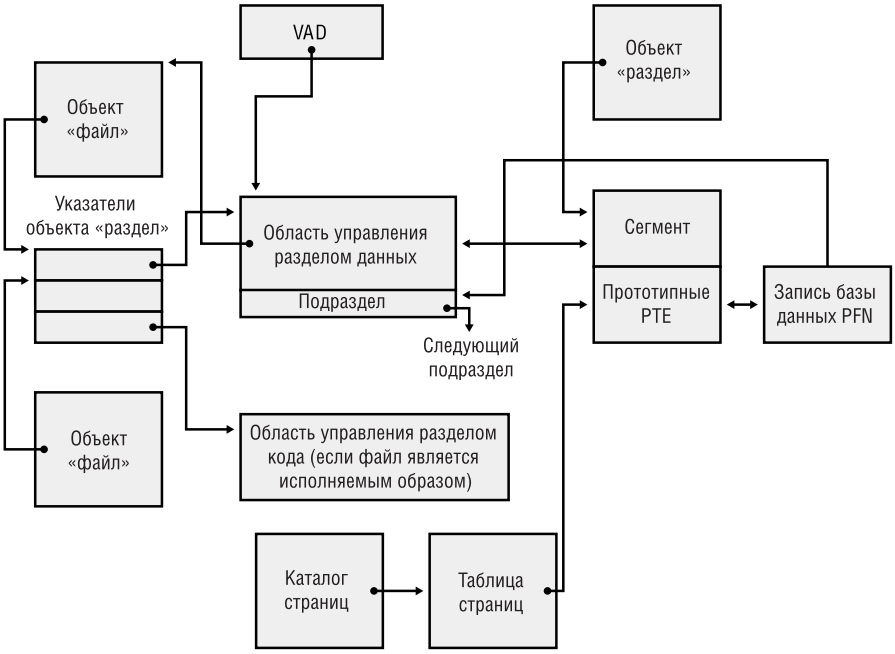


Рис. 7-30. *Внутренние структуры раздела*

Структуры данных, поддерживаемые диспетчером памяти и описывающие проецируемые разделы, показаны на рис. 7-30. Эти структуры гарантируют согласованность данных, считанных из проецируемого файла, независимо от типа доступа.

Каждому открытому файлу, представленному объектом «файл», соответствует *структура указателей объекта «раздел»* (section object pointers structure). Эта структура является ключевой для поддержания согласованности данных при всех типах доступа к файлу; она же используется и при кэшировании файлов. Структура указателей объекта «раздел» ссылается на одну или две *области управления* (control areas). Одна из них используется для проецирования файла при обращении к нему как к файлу данных, а другая — для проецирования файла при запуске его как исполняемого образа.

Область управления в свою очередь указывает на *структуры подраздела* (subsection structures), содержащие информацию о проецировании каждого раздела файла (только для чтения, для чтения и записи, копирование при записи и т. д.). Область управления также ссылается на *структуру сегмента* (segment structure), которая создается в пуле подкачиваемой памяти и указывает на прототипные PTE, указывающие на реальные страницы, проецируемые объектом «раздел». Как уже говорилось, таблицы страниц процесса ссылаются на эти прототипные PTE, а те указывают на страницы, к которым происходит обращение.

Хотя Windows гарантирует, что любой процесс, обращающийся к файлу (для чтения или записи), всегда имеет дело с согласованными данными, возможна одна ситуация, при которой в физической памяти могут находиться две копии страниц файла (но и в этом случае предоставляется только самая последняя копия и поддерживается согласованность данных). Такое дублирование происходит из-за обращения к файлу образа как к файлу данных (для чтения или записи) с его последующим запуском как исполняемого файла. Например, при сборке и последующем запуске файла образа компоновщик открывает его для доступа к данным, а при запуске программы загрузчик образов проецирует этот файл как исполняемый. При этом выполняются следующие операции.

1. Если исполняемый образ был создан через API-функции проецирования файлов (или с помощью диспетчера кэша), создается и область управления для представления считываемых или записываемых страниц данных в этом файле.
2. Когда запускается образ и создается объект «раздел» для проецирования образа как исполняемого, диспетчер памяти обнаруживает, что указатели объекта «раздел» для файла образа ссылаются на область управления данными, и сбрасывает этот раздел на диск. Эта операция нужна для того, чтобы гарантировать сохранение любых модифицированных страниц на диске до обращения к образу через область управления кодом.
3. Диспетчер памяти создает область управления кодом.

4. Как только начинается выполнение образа, обращение к страницам его файла (доступным только для чтения) вызывает ошибки страниц, и они загружаются в память.

Поскольку страницы, проецируемые областью управления данными, все еще могут быть резидентными (в списке простаивающих страниц), эта ситуация является одним из примеров существования двух копий одних и тех же данных на разных страницах памяти. Но такое дублирование не нарушает согласованность данных, поскольку область управления данными уже сброшена на диск, а значит, страницы, считанные из файла, содержат последние данные (причем эти страницы никогда не записываются обратно на диск).

ЭКСПЕРИМЕНТ: просмотр областей управления

Чтобы найти адрес структур областей управления, вы должны сначала найти адрес нужного объекта «файл». Его можно получить с помощью отладчика ядра, создав командой *!handle* дамп таблицы описателей, принадлежащей процессу. Хотя команда *!file* отладчика ядра сообщает основные сведения об объекте «файл», она не дает указатель на структуру указателей объекта «раздел». Затем, используя команду *dt*, отформатируйте объект «файл», чтобы получить адрес структуры указателей объекта «раздел». Эта структура состоит из трех указателей: на область управления данными, на разделяемую проекцию кэша (см. главу 11) и на область управления кодом. Получив адрес нужной области управления (если она есть) из структуры указателей объекта «раздел», укажите его как аргумент в команде *!ca*.

Скажем, если вы откроете файл PowerPoint и выведете таблицу описателей для этого процесса командой *!handle*, то найдете открытый описатель файла PowerPoint, как показано ниже. (Об использовании команды *!handle* см. раздел «Диспетчер объектов» главы 3.)

```
lkd> !handle 1 f 5c4 file
...
0284: Object: 8645c038  GrantedAccess: 00120089
Object: 8645c038  Type: (867ddca0) File
    ObjectHeader: 8645c020
        HandleCount: 1  PointerCount: 1
        Directory Object: 00000000  Name: \slides\ntint\new\
3-systemarchitecture.ppt {HarddiskVolume1}
```

Взяв адрес объекта «файл» (8645c038) и отформатировав его командой *dt*, вы получите:

```
lkd> dt nt!_file_object 8645c038
nt!_FILE_OBJECT
   +0x000 Type             : 5
   +0x002 Size             : 112
   +0x004 DeviceObject     : 0x86742e30
   +0x008 Vpb              : 0x867a3090
```

см. след. стр.

```
   +0x00c FsContext        : 0xe2786530
   +0x010 FsContext2       : 0xe309a378
   +0x014 SectionObjectPointer : 0x85512fec
```

Затем, сделав то же самое, но применительно к адресу структуры указателей объекта «раздел» (0x85512fec), вы получите:

```
lkd> dt nt!_section_object_pointers 0x85512fec
nt!_SECTION_OBJECT_POINTERS
   +0x000 DataSectionObject : 0x8564f8a0
   +0x004 SharedCacheMap   : (null)
   +0x008 ImageSectionObject : (null)
```

Наконец, команда *!ca* покажет вам область управления по этому адресу:

```
lkd> !ca 0x8564f8a0
ControlArea @8564f8a0
  Segment:    e3817528    Flink              0   Blink                 0
  Section Ref        1    Pfn Ref          25c   Mapped Views          1
  User Ref           2    WaitForDel         0   Flush Count           0
  File Object 85774580    ModWriteCount      0   System Views          0
  Flags (9008080) File WasPurged HadUserReference Accessed
  File: \slides\ntint\new\3-systemarchitecture.ppt
```

Другой метод — применение команды *!memusage*. Ниже приведен фрагмент вывода этой команды.

```
kd> !memusage
 loading PFN database
loading (99% complete)
             Zeroed:      9 (    36 kb)
               Free:      0 (     0 kb)
            Standby:   2103 (  8412 kb)
           Modified:    300 (  1200 kb)
    ModifiedNoWrite:      1 (     4 kb)
       Active/Valid:  30318 (121272 kb)
         Transition:      3 (    12 kb)
            Unknown:      0 (     0 kb)
              TOTAL:  32734 (130936 kb)
  Building kernel map
  Finished building kernel map
  Usage Summary (in Kb):
Control Valid Standby Dirty Shared Locked PageTables  name
8119b608  3000  528     0     0     0     0  mapped_file( WINWORD.EXE )
849e7c68    96    0     0     0     0     0    No Name for File
8109c388    24    0     0     0     0     0  mapped_file( DEVDTG.PKG )
81402488   236    0     0     0     0     0    No Name for File
80fba0a8   268    0     0     0     0     0  mapped_file( kernel32.pdb )
810ab168  1504  380     0     0     0     0  mapped_file( OUTLLIB.DLL )
81126a08     0  276     0     0     0     0  mapped_file( H )
```

```
81112d28   656    0     0     0     0     0    No Name for File
...
```

Значения в колонке Control указывают на структуру области управления, описывающую проецируемый файл. Вы можете просмотреть области управления, сегменты и подразделы с помощью команды *!ca* отладчика ядра. Например, чтобы получить дамп области управления для проецируемого файла Winword.exe, введите команду *!ca* и укажите в ней число из колонки Control, как показано ниже.

```
kd> !ca 8119b608

ControlArea @8119b608
  Segment:    e2c28000  Flink             0  Blink:            0
  Section Ref        1  Pfn Ref         372  Mapped Views:     1
  User Ref           2  Subsections       6  Flush Count:      0
  File Object 8213fb98  ModWriteCount     0  System Views:     0
  WaitForDel         0  Paged Usage    3000  NonPaged Usage  100
  Flags (90000a0) Image File HadUserReference Accessed

   File: \Program Files\Microsoft Office\Office\WINWORD.EXE

Segment @ e2c28000:
   Base address        0  Total Ptes       86a  NonExtendPtes:      86a
   Image commit       d7  ControlArea 8119b608  SizeOfSegment: 86a000
   Image Base          0  Committed          0  PTE Template:  919b6c38
   Based Addr   30000000  ProtoPtes   e2c28038  Image Info:    e2c2a1e4

Subsection 1. @ 8119b640
   ControlArea: 8119b608  Starting Sector 0 Number Of Sectors 8
   Base Pte     e2c28038  Ptes In subsect        1 Unused Ptes          0
   Flags              15  Sector Offset          0 Protection           1
    ReadOnly CopyOnWrite

Subsection 2. @ 8119b660
   ControlArea: 8119b608  Starting Sector 10 Number Of Sectors 3c00
   Base Pte     e2c2803c  Ptes In subsect      780 Unused Ptes          0
   Flags              35  Sector Offset          0 Protection           3
    ReadOnly CopyOnWrite

Subsection 3. @ 8119b680
   ControlArea: 8119b608  Starting Sector 3C10 Number Of Sectors 5d8
   Base Pte     e2c29e3c  Ptes In subsect       c1 Unused Ptes          0
   Flags              55  Sector Offset          0 Protection           5
    ReadOnly CopyOnWrite

Subsection 4. @ 8119b6a0
   ControlArea: 8119b608  Starting Sector 41E8 Number Of Sectors a8
```

см. след. стр.

```
    Base Pte     e2c2a140  Ptes In subsect       15 Unused Ptes          0
    Flags              55  Sector Offset          0 Protection           5
     ReadOnly CopyOnWrite

Subsection 5. @ 8119b6c0
    ControlArea: 8119b608  Starting Sector 0 Number Of Sectors 0
    Base Pte     e2c2a194  Ptes In subsect        1 Unused Ptes          0
    Flags              55  Sector Offset          0 Protection           5
     ReadOnly CopyOnWrite

Subsection 6. @ 8119b6e0
    ControlArea: 8119b608  Starting Sector 4290 Number Of Sectors 90
    Base Pte     e2c2a198  Ptes In subsect       12 Unused Ptes          0
    Flags              15  Sector Offset          0 Protection           1
     ReadOnly CopyOnWrite
```

Рабочие наборы

Здесь мы сосредоточимся на виртуальной части Windows-процесса — таблицах страниц, PTE и VAD. В оставшейся части главы мы расскажем, как Windows хранит в физической памяти подмножество виртуальных адресов.

Как вы помните, подмножество виртуальных страниц, резидентных в физической памяти, называется *рабочим набором* (working set). Существует три вида рабочих наборов:

- процесса — содержит страницы, на которые ссылаются его потоки;
- системы — содержит резидентное подмножество подкачиваемого системного кода (например, Ntoskrnl.exe и драйверов), пула подкачиваемой памяти и системного кэша;
- сеанса — в системах с включенной службой Terminal Services каждый сеанс получает свой рабочий набор. Он содержит резидентное подмножество специфичных для сеанса структур данных режима ядра, создаваемых частью подсистемы Windows, которая работает в режиме ядра (Win32k.sys), пула подкачиваемой памяти сеанса, представлений, проецируемых в сеансе, и других драйверов устройств, проецируемых на пространство сеанса.

Прежде чем детально рассматривать каждый тип рабочего набора, обсудим общие правила выбора страниц, загружаемых в память, и определения срока их пребывания в физической памяти.

Подкачка по требованию

Диспетчер памяти Windows использует алгоритм подкачки по требованию с кластеризацией. Когда поток вызывает ошибку страницы, диспетчер памяти загружает не только страницу, при обращении к которой возникла ошибка, но и несколько предшествующих и/или последующих страниц. Эта стратегия обеспечивает минимизацию числа операций ввода-вывода, связанных с подкачкой. Поскольку программы (особенно большие) в любой момент

времени обычно выполняются в небольших областях своего адресного пространства, загрузка виртуальных страниц кластерами уменьшает число операций чтения с диска. При ошибках, связанных со ссылками на страницы данных в образах, размер кластера равен 3, в остальных случаях — 7.

Однако политика подкачки по требованию может привести к тому, что какой-то процесс будет вызывать очень много ошибок страниц в момент начала выполнения его потоков или позднее при возобновлении их выполнения. Для оптимизации запуска процесса (и системы) в Windows XP и Windows Server 2003 введен механизм интеллектуальной предвыборки (intelligent prefetch engine), также называемый средством логической предвыборки (logical prefetcher); о нем мы рассказываем в следующем разделе.

Средство логической предвыборки

В ходе типичной загрузки системы или приложения порядок ошибок страниц таков, что некоторые страницы запрашиваются из одной части файла, затем из совсем другой его части, потом из другого файла и т. д. Такие скачкообразные переходы значительно замедляют каждую операцию доступа, и, как показывает анализ, время поиска на диске становится доминирующим фактором, который негативно сказывается на скорости загрузки системы и приложений. Предварительная выборка целого пакета страниц позволяет упорядочить операции доступа без лишнего «рыскания» по диску и тем самым ускорить запуск системы и приложений. Нужные страницы могут быть известны заранее благодаря высокой степени корреляции операций доступа при загрузках системы или запусках приложений.

Средство предвыборки, впервые появившееся в Windows XP, пытается ускорить загрузку системы и запуск приложений, отслеживая данные и код, к которым происходит обращение при этих процессах, и используя полученную информацию при последующих загрузке системы и запуске приложений для заблаговременного считывания необходимых кода и данных. Когда средство предвыборки активно, диспетчер памяти уведомляет код средства предвыборки в ядре об ошибках страниц — как аппаратных (требующих чтения данных с диска), так и программных (требующих простого добавления данных, которые уже находятся в памяти, в рабочий набор процесса). Средство предвыборки ведет мониторинг первых 10 секунд процесса запуска приложения. В случае загрузки системы это средство по умолчанию ведет мониторинг в течение 30 секунд после запуска пользовательской оболочки (обычно Explorer), или 60 секунд по окончании инициализации всех Windows-сервисов, или просто в течение 120 секунд — в зависимости от того, какое из этих трех событий произойдет первым.

Собрав трассировочную информацию об ошибках страниц, организованную в виде списка обращений к файлу метаданных NTFS MFT (Master File Table) (если приложение пыталось получить доступ к файлам или каталогам на NTFS-томах), а также списка ссылок на файлы и каталоги, код средства предвыборки, работающий в режиме ядра, уведомляет компонент предвыборки в службе Task Scheduler (Планировщик заданий) (\Windows\System32\

Schedsvc.dll) и с этой целью переводит в свободное состояние объект-событие с именем *PrefetchTracesReady*.

> **ПРИМЕЧАНИЕ** Включить или отключить предвыборку при загрузке системы и/или запуске приложений позволяет DWORD-параметр реестра HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management\PrefetchParameters\EnablePrefetcher: 0 — полное отключение предвыборки, 1 — предвыборка разрешена только при запуске приложений, 2 — предвыборка разрешена только при загрузке системы и 3 — предвыборка разрешена как при загрузке системы, так и при запуске приложений.

Когда событие *PrefetchTracesReady* переводится в свободное состояние, Task Scheduler вызывает внутрисистемную функцию *NtQuerySystemInformation*, запрашивая через нее трассировочные данные. После дополнительной обработки этих данных Task Scheduler записывает их в файл, помещаемый в каталог \Windows\Prefetch (рис. 7-31). Файлу присваивается имя, формируемое из имени приложения, к которому относятся трассировочные данные, дефиса и шестнадцатеричного представления хэша, полученного из строки пути к файлу. Затем к имени добавляется расширение .pf. Например, для Notepad создается файл NOTEPAD.EXE-AF43252301.PF.

В этом правиле есть два исключения. Первое относится к образам, которые служат хостами других компонентов, в том числе к Microsoft Management Console (\Windows\System32\Mmc.exe) и Dllhost (\Windows\System32\Dllhost.exe). Поскольку в командной строке запуска этих приложений указываются дополнительные компоненты, средство предвыборки включает командную строку в генерируемый хэш. Таким образом, запуск таких приложений с другими компонентами в командной строке даст другой набор трассировочных данных. Средство предвыборки считывает список исполняемых образов, которые оно должно обрабатывать таким способом, из параметра HostingAppList в разделе реестра HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management\PrefetchParameters.

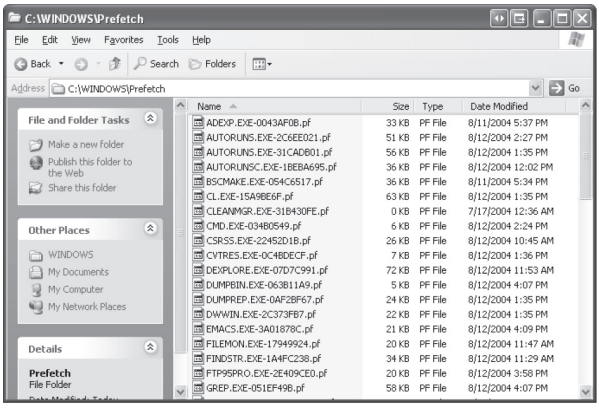


Рис. 7-31. *Каталог Prefetch*

Второе исключение составляет файл, в котором хранится трассировочная информация, полученная в процессе загрузки системы, — ему всегда присваивается имя NTOSBOOT-B00DFAAD.PF. Средство предвыборки собирает информацию об ошибках страниц для конкретных приложений только после того, как закончит мониторинг процесса загрузки системы.

ЭКСПЕРИМЕНТ: просмотр содержимого файла предвыборки

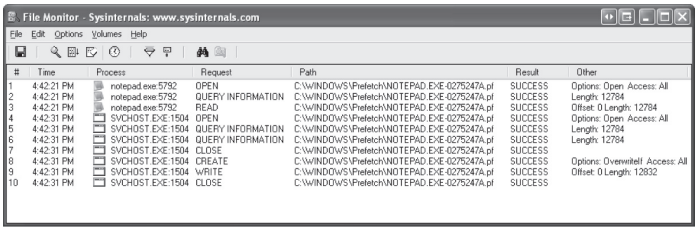
Содержимое этого файла содержит записи о файлах и каталогах, к которым было обращение при загрузке системы или запуске приложения, и для их просмотра можно использовать утилиту Strings с сайта *www.sysinternals.com*. Следующая команда перечисляет все файлы и каталоги, на которые были ссылки при последней загрузке системы:

```
C:\Windows\Prefetch>Strings ntosboot-boodfaad.pf

Strings v2.1
Copyright (C) 1999-2003 Mark Russinovich
Systems Internals - www.sysinternals.com

NTOSBOOT
SCCA
\DEVICE\HARDDISKVOLUME2\$MFT
\DEVICE\HARDDISKVOLUME2\WINDOWS\PREFETCH\NTOSBOOT-B00DFAAD.PF
\DEVICE\HARDDISKVOLUME2\SYSTEM VOLUME INFORMATION\_RESTORE{
987E0331-0F01-427C-A58A-7A2E4AABF84D}\RP24\CHANGE.LOG
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\PROCESSR.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\FGLRYM.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\VIDEOPRT.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\E1000325.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\USBUHCI.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\USBPORT.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\USBEHCI.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\NIC1394.SYS
...
```

Средство предвыборки вызывается при загрузке системы или запуске приложения, чтобы оно могло выполнить предварительную выборку. Средство предвыборки просматривает каталог Prefetch и проверяет, есть ли в нем какой-нибудь файл с трассировочной информацией, необходимый для текущего варианта предварительной выборки. Если такой файл имеется, оно обращается к NTFS для предварительной выборки любых ссылок файла метаданных MFT, считывает содержимое каждого каталога, на который есть ссылка, а затем открывает все файлы в соответствии со списком ссылок. Далее вызывается функция *MmPrefetchPages* диспетчера памяти, чтобы загрузить в память любые данные и код, указанные в трассировочной информации, но пока отсутствующие в памяти. Диспетчер памяти инициирует все необходимые операции как асинхронные и ждет их завершения, прежде чем разрешить продолжение процесса запуска приложения.

ЭКСПЕРИМЕНТ: наблюдение за чтением и записью файла предвыборки

Если вы запишете трассировку запуска приложения с помощью Filemon (*www.sysinternals.com*) в Windows XP, то заметите, что средство предвыборки проверяет наличие файла предвыборки и, если он есть, считывает его содержимое, а примерно через десять секунд от начала запуска приложения средство предвыборки записывает новую копию этого файла. Ниже показан пример для процесса запуска Notepad (фильтр Include был установлен как «prefetch», чтобы Filemon сообщал об обращениях только к каталогу \Windows\Prefetch).



Строки 1–3 показывают, что файл предвыборки Notepad считывался в контексте процесса Notepad в ходе его запуска. Строки 4–10 (с временными метками на 10 секунд позже, чем в первых трех строках) демонстрируют, что Task Scheduler, который выполняется в контексте процесса Svchost, записал обновленный файл предвыборки.

Чтобы еще больше уменьшить вероятность скачкообразного поиска, через каждые три дня (или около того) Task Scheduler в периоды простоя формирует список файлов и каталогов в том порядке, в каком на них были ссылки при загрузке системы или запуске приложения, и сохраняет его в файле \Windows\Prefetch\Layout.ini (рис. 7-32).

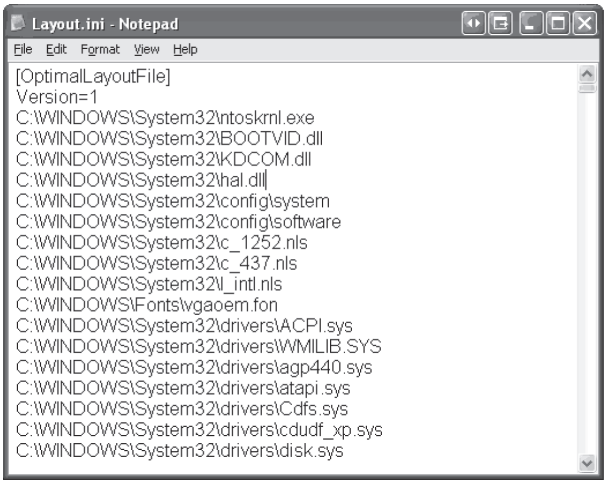


Рис. 7-32. *Содержимое файла Layout.ini*

Далее он запускает системный дефрагментатор, указывая ему через командную строку выполнить дефрагментацию на основе содержимого файла Layout.ini. Дефрагментатор находит на каждом томе непрерывную область, достаточно большую, чтобы в ней уместились все файлы и каталоги, перечисленные для данного тома, а затем целиком перемещает их в эту область в указанном порядке. Благодаря этому будущие операции предварительной выборки окажутся еще эффективнее, поскольку все считываемые данные теперь физически хранятся на диске в нужной последовательности. Такая дефрагментация обычно затрагивает всего несколько сотен файлов и поэтому выполняется гораздо быстрее, чем полная дефрагментация диска. (Подробнее о дефрагментации см. в главе 12.)

Правила размещения

Когда поток вызывает ошибку страницы, диспетчер памяти должен также определить, в каком участке физической памяти следует разместить виртуальную страницу. При этом он руководствуется *правилами размещения* (placement policy). Выбирая фреймы страниц, Windows учитывает размер кэшей процессора и стремится свести нагрузку на них к минимуму.

Если на момент появления ошибки страницы физическая память заполнена, выбирается страница, подлежащая выгрузке на диск для освобождения памяти под новую страницу. Этот выбор осуществляется по *правилам замены* (replacement policy). При этом действуют два общих правила замены: LRU (least recently used) и FIFO (first in, first out). Алгоритм LRU (также известный как алгоритм часов и реализованный в большинстве версий UNIX) требует от подсистемы виртуальной памяти следить за тем, когда используется страница в памяти. Страница, не использовавшаяся в течение самого длительного времени, удаляется из рабочего набора. Алгоритм FIFO работает проще: он выгружает из физической памяти страницу, которая находилась там дольше всего независимо от частоты ее использования.

Правила замены страниц могут быть глобальными или локальными. Глобальные правила позволяют использовать для обработки ошибки страницы любой фрейм страниц независимо от того, принадлежит ли он другому процессу. Например, в результате применения глобальных правил замены с применением алгоритма FIFO будет найдена и выгружена на диск страница, находившаяся в памяти наибольшее время, а локальные правила замены ограничат сферу поиска самой старой страницей из набора, который принадлежит процессу, вызвавшему ошибку страницы. Таким образом, глобальные правила замены делают процессы уязвимыми от поведения других процессов, и одно сбойное приложение может негативно отразиться на всей операционной системе.

В Windows реализована комбинация локальных и глобальных правил замены. Когда размер рабочего набора достигает своего лимита и/или появляется необходимость его усечения из-за нехватки физической памяти, диспетчер памяти удаляет из рабочих наборов ровно столько страниц, сколько ему нужно освободить.

Управление рабочими наборами

Все процессы начинают свой жизненный цикл с максимальным и минимальным размерами рабочего набора по умолчанию — 50 и 345 страниц соответственно. Хотя это мало что дает, эти значения по умолчанию можно изменить для конкретного процесса через Windows-функцию *SetProcessWorkingSetSize*, но для этого нужна привилегия Increase Scheduling Priority. Однако, если только вы не укажете процессу использовать жесткие лимиты на рабочий набор (новшество Windows Server 2003), эти лимиты игнорируются в том смысле, что диспетчер памяти разрешит процессу расширение за установленный максимум при наличии интенсивной подкачки страниц и достаточного объема свободной памяти (либо, напротив, уменьшит его рабочий набор ниже минимума при отсутствии подкачки страниц и при высокой потребности системы в физической памяти). Хотя в Windows 2000 степень расширения процесса за максимальную границу рабочего набора увязывалась с вероятностью его усечения, в Windows XP это решение принимается исключительно на основе того, к скольким страницам обращался процесс.

В Windows Server 2003 жесткие лимиты на размеры рабочего набора могут быть заданы вызовом функции *SetProcessWorkingSetSizeEx* с флагом QUOTA_LIMITS_HARDWS_ENABLE. Этой функцией пользуется, например, Windows System Resource Manager (WSRM), описанный в главе 6.

Максимальный размер рабочего набора не может превышать общесистемный максимум, вычисленный при инициализации системы и хранящийся в переменной ядра *MmMaximumWorkingSetSize*. Это значение представляет собой число страниц, доступных на момент вычислений (суммарный размер списков обнуленных, свободных и простаивающих страниц), за вычетом 512 страниц. Однако существуют жесткие верхние лимиты на размеры рабочих наборов — они перечислены в таблице 7-17.

Таблица 7-17. *Верхний лимит на максимальные размеры рабочих наборов*

Версия Windows	Максимальный размер рабочего набора
x86-версии Windows 2000, Windows XP, Windows XP SP1, Windows Server 2003	1984 Мб
x86-версии Windows XP SP2, Windows Server 2003 SP1	2047,9 Мб
x86-версии Windows, загружаемые с ключом /3GB	3008 Мб
IA64	7152 Гб
x64	8192 Гб

Когда возникает ошибка страницы, система проверяет лимиты рабочего набора процесса и объем свободной памяти. Если условия позволяют, диспетчер памяти разрешает процессу увеличить размер своего рабочего набора до максимума (и даже превысить его, если свободных страниц достаточно и если для этого процесса не задан жесткий лимит на размер рабочего набора). Но если памяти мало, Windows предпочитает заменять страницы в рабочем наборе, а не добавлять в него новые.

Хотя Windows пытается поддерживать достаточный объем доступной памяти, записывая измененные страницы на диск, при слишком быстрой генерации модифицированных страниц понадобится больше свободной памяти. Поэтому, когда свободной физической памяти становится мало, вызывается *диспетчер рабочих наборов* (working set manager), который выполняется в контексте системного потока диспетчера настройки баланса (см. следующий раздел) и инициирует автоматическое усечение рабочего набора для увеличения объема доступной в системе свободной памяти. (Используя Windows-функцию *SetProcessWorkingSetSize*, вы можете вызвать усечение рабочего набора своего процесса, например после его инициализации.)

Диспетчер рабочих наборов принимает решения об усечении каких-либо рабочих наборов, исходя из объема доступной памяти. Если памяти достаточно, он подсчитывает, сколько страниц можно при необходимости изъять из рабочего набора. Как только такая необходимость появится, он уменьшит рабочие наборы, размер которых превышает минимальный. Диспетчер рабочих наборов динамически регулирует частоту проверки рабочих наборов и оптимальным образом упорядочивает список процессов — кандидатов на усечение рабочего набора. Например, первыми проверяются процессы со множеством страниц, к которым не было недавних обращений; часто простаивающие процессы большего размера являются более вероятными кандидатами, чем реже простаивающие процессы меньшего размера; процессы активного приложения рассматриваются в последнюю очередь и т. д.

Определив, что размеры рабочих наборов процессов превышают минимальные значения, диспетчер ищет страницы, которые можно удалить из их рабочих наборов и сделать доступными для использования в других целях. Если свободной памяти по-прежнему не хватает, диспетчер продолжает удалять страницы из рабочих наборов процессов, пока в системе не появится минимальное количество свободных страниц.

В однопроцессорных системах Windows 2000 и всех системах Windows XP или Windows Server 2003 диспетчер рабочих наборов старается удалять страницы, к которым не было обращений в последнее время. Такие страницы обнаруживаются проверкой битового флага Accessed в аппаратном PTE. Если этот флаг сброшен, страница считается устаревшей, и соответствующий счетчик увеличивается на 1, показывая, что с момента последнего сканирования данного рабочего набора к этой странице не было обращений. Впоследствии возраст страниц учитывается при поиске кандидатов на удаление из рабочего набора.

ПРИМЕЧАНИЕ В многопроцессорной системе Windows 2000 диспетчер рабочего набора ошибочно не проверял битовый флаг Accessed, что приводило к удалению страниц из рабочего набора без учета состояния этого флага.

Если битовый флаг Accessed в аппаратном PTE установлен, диспетчер рабочих наборов сбрасывает его и проверяет следующую страницу рабочего набора. Таким образом, если при следующем сканировании битовый флаг

Accessed окажется сброшенным, диспетчер будет знать, что со времени последнего сканирования к этой странице не было обращений. Сканирование списка рабочего набора продолжается до удаления нужного числа страниц или до возврата к начальной точке. (В следующий раз сканирование начнется там, где оно остановилось в прошлый раз.)

ЭКСПЕРИМЕНТ: просмотр размеров рабочих наборов процессов

С этой целью можно использовать счетчики в оснастке Performance (Производительность), перечисленные в следующей таблице.

Счетчик	Описание
Process: Working Set (Процесс: Рабочее множество)	Текущий размер рабочего набора выбранного процесса (в байтах)
Process: Working Set Peak [Процесс: Рабочее множество (пик)]	Пиковый размер рабочего набора выбранного процесса (в байтах)
Process: Page Faults/Sec (Процесс: Ошибок страницы/сек)	Число ошибок страниц, генерируемых процессом каждую секунду

Несколько других утилит для просмотра сведений о процессах (Task Manager, Pview и Pviewer) тоже показывают размеры рабочих наборов.

Суммарный размер рабочих наборов всех процессов можно получить, выбрав процесс _Total в оснастке Performance. Этот несуществующий процесс просто представляет суммарные значения счетчиков всех процессов, выполняемых в системе в данный момент. Однако суммарный размер не соответствует истине, так как при подсчете размера рабочего набора процесса учитываются его разделяемые страницы. В итоге страница, разделяемая двумя или более процессами, засчитывается в размер рабочего набора каждого из таких процессов.

ЭКСПЕРИМЕНТ: просмотр списка рабочего набора

Элементы рабочего набора можно увидеть с помощью команды *!wsle* отладчика ядра. Ниже показан фрагмент выходной информации о списке рабочего набора, полученной с помощью LiveKd (эта команда была выполнена применительно к процессу LiveKd).

```
kd> !wsle 7

Working Set @ c0502000
    Quota:      9f  FirstFree:  40  FirstDynamic:      3
    LastEntry  1fe  NextSlot:    3  LastInitialized  257
    NonDirect   5c  HashTable:   0  HashTableSize:     0

Virtual Address  Age  Locked  ReferenceCount
c0300203          0       1       1
c0301203          0       1       1
c0502203          0       1       1
```

```
c01df201          0          0          1
c01ff201          0          0          1
c0005201          0          0          1
c0001201          0          0          1
c0002201          0          0          1
c0000201          0          0          1
c0006201          0          0          1
77e87119          0          0          1
00402319          0          0          1
77e01201          0          0          1
7ffdf201          0          0          1
00130201          0          0          1
77e9e119          0          0          1
78033201          0          0          1
00230221          0          0          1
00131201          0          0          1
77d50119          0          0          1
00132201          0          0          1
c01e0201          0          0          1
00411309          0          0          1
0040d201          0          0          1
77edf201          0          0          1
77ee0201          0          0          1
77fcd201          0          0          1
0040e201          0          0          1
7ffc1009          0          0          1
00401319          0          0          1
```

Заметьте, что одни элементы списка рабочего набора представляют собой страницы, содержащие таблицы страниц (элементы с адресами выше 0xC0000000), другие — страницы системных DLL (в диапазоне 0x7*nnnnnnn*), третьи — страницы кода самой LiveKd.exe (в диапазоне 0x004*nnnnn*).

Диспетчер настройки баланса и подсистема загрузки-выгрузки

Расширение и усечение рабочего набора выполняется в контексте системного потока диспетчера настройки баланса (balance set manager) (процедура *KeBalanceSetManager*). Его поток создается при инициализации системы. Хотя с технической точки зрения диспетчер настройки баланса входит в состав ядра, для анализа и регулировки рабочих наборов он обращается к диспетчеру рабочих наборов.

Диспетчер настройки баланса ждет на двух объектах «событие»: один из них освобождается по сигналу таймера, срабатывающего раз в секунду, а другой представляет собой внутреннее событие диспетчера рабочих наборов, освобождаемое диспетчером памяти, когда возникает необходимость в изме-

нении рабочих наборов. Например, если в системе слишком часто генерируются ошибки страниц или список свободных страниц слишком мал, диспетчер памяти пробуждает диспетчер настройки баланса, а тот вызывает диспетчер рабочих наборов для усечения таких наборов. Если свободной памяти много, диспетчер рабочих наборов разрешает процессам, часто вызывающим ошибки страниц, постепенно увеличивать размеры своих рабочих наборов, подкачивая в память страницы, при обращении к которым возникали ошибки. Однако рабочие наборы расширяются лишь по мере необходимости.

Диспетчер настройки баланса, пробуждаемый в результате срабатывания таймера, выполняет следующие операции.

1. При каждом четвертом пробуждении из-за срабатывания таймера освобождает событие, которое активизирует системный поток, выполняющий процедуру *KeSwapProcessOrStack* — подсистему загрузки-выгрузки (swapper).
2. Проверяет ассоциативные списки и регулирует глубину их вложения, если это необходимо (для ускорения доступа, снижения нагрузки на пул и уменьшения его фрагментации).
3. Ищет потоки, чей приоритет может быть повышен из-за нехватки процессорного времени (см. раздел «Динамическое повышение приоритета при нехватке процессорного времени» главы 6).
4. Вызывает диспетчер рабочих наборов (имеющий собственные внутренние счетчики, которые определяют, когда и насколько агрессивно следует проводить усечение рабочих наборов).

Подсистема загрузки-выгрузки пробуждается и планировщиком, если стек ядра потока, подлежащего выполнению, или весь процесс выгружен в страничный файл. Подсистема загрузки-выгрузки ищет потоки, которые находились в состоянии ожидания в течение 7 секунд (Windows 2000) или 15 секунд (Windows XP и Windows Server 2003). Если такой поток есть, подсистема загрузки-выгрузки переводит его стек ядра в переходное состояние (перемещая соответствующие страницы в список модифицированных или простаивающих страниц). Здесь действует принцип «если поток столько времени ждал, подождет и еще». Когда из памяти удаляется стек ядра последнего потока процесса, этот процесс помечается как полностью выгруженный. Вот почему у долго простаивавших процессов (например, у Winlogon после вашего входа в систему) может быть нулевой размер рабочих наборов.

Системный рабочий набор

Подкачиваемый код и данные операционной системы тоже управляются как единый *системный рабочий набор* (system working set). В системный рабочий набор могут входить страницы пяти видов:

- системного кэша;
- пула подкачиваемой памяти;
- подкачиваемого кода и данных Ntoskrnl.exe;
- подкачиваемого кода и данных драйверов устройств;

- проецируемых системой представлений.

Выяснить размер системного рабочего набора и пяти его элементов можно с помощью счетчиков производительности или системных переменных, перечисленных в таблице 7-18. Учтите, что значения счетчиков выражаются в байтах, а значения системных переменных — в страницах.

Узнать интенсивность подкачки страниц в системном рабочем наборе позволяет счетчик Memory: Cache Faults/Sec (Память: Ошибок кэш-памяти/сек), который сообщает число ошибок страниц, генерируемых в системном рабочем наборе (как аппаратных, так и программных).

Таблица 7-18. *Счетчики, относящиеся к системному рабочему набору*

Счетчик	Системная переменная	Описание
Memory: Cache Bytes (Память: Байт кэш-памяти)[1]	*MmSystemCacheWs.WorkingSetSize*	Суммарный размер системного рабочего набора (включая кэш, пул подкачиваемой памяти, подкачиваемый код Ntoskrnl и драйверов, а также представления, проецируемые системой); это *не* размер только системного кэша, как можно было бы подумать по названию счетчика
Memory: Cache Bytes Peak [Память: Байт кэш-памяти (пик)]	*MmSystemCacheWs.Peak*	Пиковый размер системного рабочего набора
Memory: System Cache Resident Bytes (Память: Резидентных байт системного кэша)	*MmSystemCachePage*	Физическая память, занятая системным кэшем
Memory: System Code Resident Bytes (Память: Резидентных байт системного кода)	*MmSystemCodePage*	Физическая память, занятая подкачиваемым кодом Ntoskrnl.exe
Memory: System Driver Resident Bytes (Память: Резидентных байт системных драйверов)	*MmSystemDriverPage*	Физическая память, занятая подкачиваемым кодом драйверов устройств
Memory: Pool Paged Resident Bytes (Память: Байт в резидентном страничном пуле)	*MmPagedPoolPage*	Физическая память, занятая пулом подкачиваемой памяти

[1] Внутреннее название этого рабочего набора — рабочий набор системного кэша, хотя системный кэш является лишь одним из пяти элементов системного рабочего набора. Из-за этой путаницы некоторые утилиты, сообщая размер файлового кэша, на самом деле показывают суммарный размер системного рабочего набора.

Минимальный и максимальный размеры системного рабочего набора вычисляются при инициализации системы, исходя из объема физической памяти компьютера и выпуска Windows — клиентского или серверного.

Вычисленные значения максимального и минимального размеров хранятся в системных переменных, показанных в таблице 7-19 (их значения недоступны через счетчики производительности, но вы можете просмотреть их в отладчике ядра).

Таблица 7-19. *Системные переменные, хранящие максимальный и минимальный размеры системного рабочего набора*

Переменная	Тип	Описание
MmSystemCacheWsMinimum-WorkingSetSize	ULONG	Минимальный размер системного рабочего набора
MmSystemCacheWsMaximum-WorkingSetSize	ULONG	Максимальный размер системного рабочего набора

Вы можете отдать приоритет системному рабочему набору (в противоположность рабочим наборам процессов), изменив параметр реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\LargeSystemCache. В системах Windows 2000 Server это значение можно было косвенно модифицировать заданием свойств для службы файлового сервера; Windows XP и Windows Server 2003 позволяют сделать это явно: щелкните My Computer (Мой компьютер) правой кнопкой мыши, выберите Properties (Свойства), откройте вкладку Advanced (Дополнительно), нажмите кнопку Settings (Параметры) в разделе Performance (Быстродействие) и перейдите на очередную вкладку Advanced (детали см. в главе 11).

База данных PFN

Если рабочие наборы описывают резидентные страницы, принадлежащие процессу или системе, то *база данных PFN* (номеров фреймов страниц) определяет состояние каждой страницы в физической памяти. Состояния страниц перечислены в таблице 7-20.

Таблица 7-20. *Возможные состояния страниц*

Состояние	Описание
Активная, или действительная (active/valid)	Данная страница либо является частью рабочего набора (процесса или системы), либо не входит ни в один рабочий набор (находясь, например, в пуле неподкачиваемой памяти ядра), но при этом на нее ссылается действительный PTE
Переходная (transition)	Временное состояние страницы, не принадлежащей ни одному рабочему набору или списку. Страница пребывает в этом состоянии в ходе операции ввода-вывода. PTE программируются так, чтобы можно было обнаруживать и корректно обрабатывать конфликты ошибок страницы. (Заметьте, что здесь термин «переходная» имеет другой смысл в отличие от того же термина, употреблявшегося в разделе по недействительным PTE; недействительный переходный PTE ссылается на страницу в списке простаивающих или модифицированных страниц.)

Таблица 7-20. (*окончание*)

Состояние	Описание
Простаивающая (standby)	Данная страница раньше входила в рабочий набор, но теперь удалена из него. Страница осталась неизменной с момента последней записи на диск. PTE все еще ссылается на физическую страницу, но уже помечен как недействительный и находящийся в переходном состоянии
Модифицированная (modified)	Данная страница раньше входила в рабочий набор, но теперь удалена из него. Однако она была изменена, и ее содержимое еще не записано на диск. PTE все еще ссылается на физическую страницу, но уже помечен как недействительный и находящийся в переходном состоянии. Перед повторным использованием страницы ее содержимое должно быть записано на диск
Модифицированная, но не записываемая (modified no-write)	То же, что и предыдущее состояние, но страница помечена так, что подсистема записи модифицированных страниц (принадлежащая диспетчеру памяти) не будет записывать ее на диск. Диспетчер кэша помечает страницы как модифицированные, но не записываемые по запросу драйверов файловой системы. Так, NTFS использует это состояние для страниц, содержащих метаданные файловой системы, чтобы записи журнала транзакций сбрасывались на диск до содержимого защищаемых ими страниц (о протоколировании транзакций NTFS см. главу 12)
Свободная (free)	Эта страница свободна, но содержит какие-то данные. Ее нельзя передать пользовательскому процессу до заполнения нулями (из соображений безопасности)
Обнуленная (zeroed)	Данная страница свободна и инициализирована нулевыми значениями (потоком обнуления страниц)
Только для чтения (ROM)	Ошибка страницы была вызвана из памяти только для чтения (новшество Windows XP)
Аварийная (bad)	Данная страница вызвала ошибку четности или другую аппаратную ошибку. Ее больше нельзя использовать

База данных PFN состоит из массива структур, представляющих каждую страницу физической памяти в системе. Как показано на рис. 7-33, действительные PTE ссылаются на записи базы данных PFN, а эти записи (если они не относятся к прототипным PFN) — на таблицу страниц, которая их использует. Прототипные PFN ссылаются на прототипные PTE.

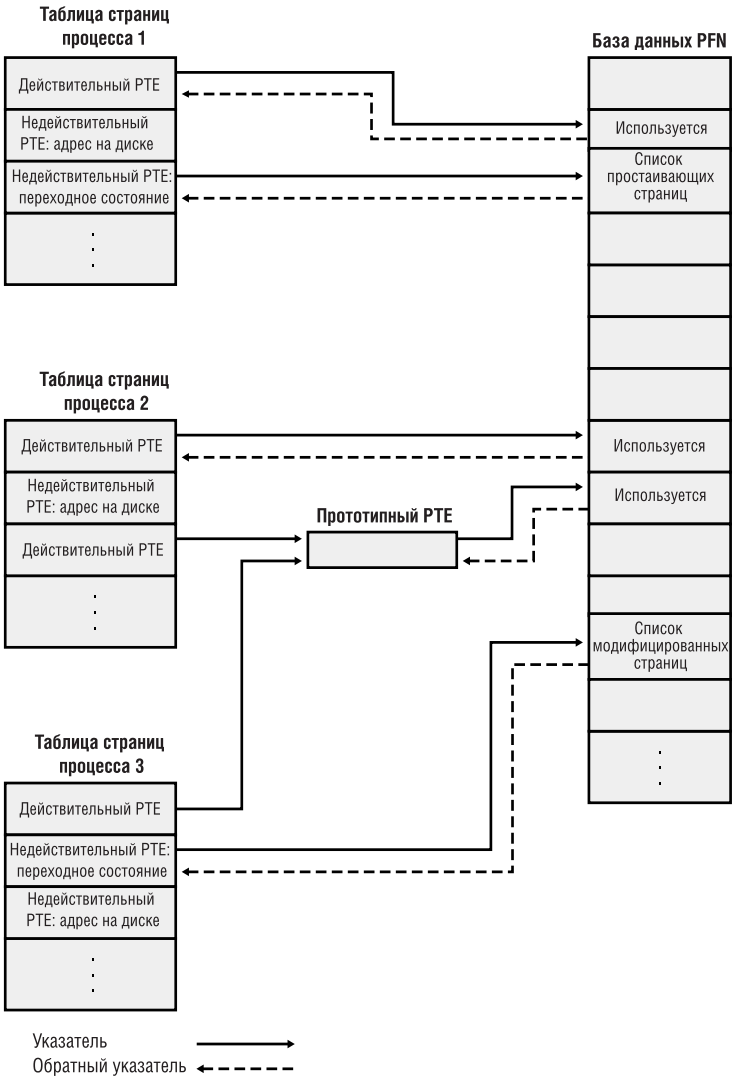


Рис. 7-33. *Таблицы страниц и база данных номеров фреймов страниц*

Страницы, находящиеся в некоторых из перечисленных в таблице 7-20 состояний, организуются в связанные списки, что помогает диспетчеру памяти быстро находить страницы определенного типа. (Активные и переходные страницы не включаются ни в один общесистемный список.) Пример взаимосвязей элементов таких списков в базе данных PFN показан на рис. 7-34.

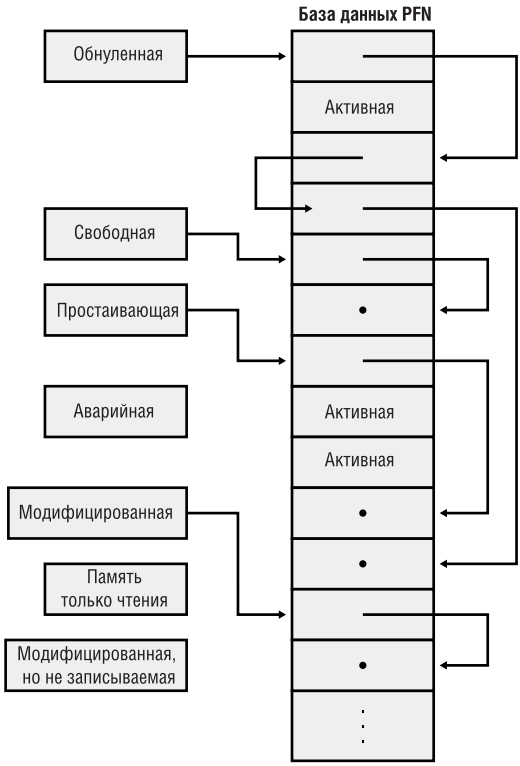


Рис. 7-34. *Списки страниц в базе данных PFN*

В следующем разделе вы узнаете, как эти связанные списки используются при обработке ошибок страниц и как страницы перемещаются между различными списками.

ЭКСПЕРИМЕНТ: просмотр базы данных PFN

Команда *!memusage* отладчика ядра позволяет получить информацию о размерах различных списков страниц. Вот пример вывода этой команды.

```
lkd> !memusage
 loading PFN database
loading (100% complete)
Compiling memory usage data (99% Complete).
              Zeroed:   8474 (33896 kb)
                Free:    256 (  1024 kb)
             Standby:  50790 (203160 kb)
            Modified:    496 (  1984 kb)
    ModifiedNoWrite:      0 (     0 kb)
       Active/Valid: 201980 (807920 kb)
         Transition:      1 (     4 kb)
            Unknown:      0 (     0 kb)
              TOTAL: 261997 (1047988 kb)
```

Динамика списков страниц

На рис. 7-35 показана схема состояний фрейма страниц. Для упрощения на ней отсутствует список модифицированных, но не записываемых страниц.

Фреймы страниц перемещаются между различными списками следующим образом.

- Когда диспетчеру памяти нужна обнуленная страница для обслуживания ошибки страницы, обнуляемой по требованию (demand-zero page fault) (ссылки на страницу, которая должна быть заполнена нулями, или на закрытую переданную страницу пользовательского режима, к которой еще не было обращений), он прежде всего пытается получить ее из списка обнуленных страниц. Если этот список пуст, он берет ее из списка свободных страниц и заполняет нулями. Если и этот список пуст, диспетчер памяти извлекает страницу из списка простаивающих страниц и обнуляет ее.

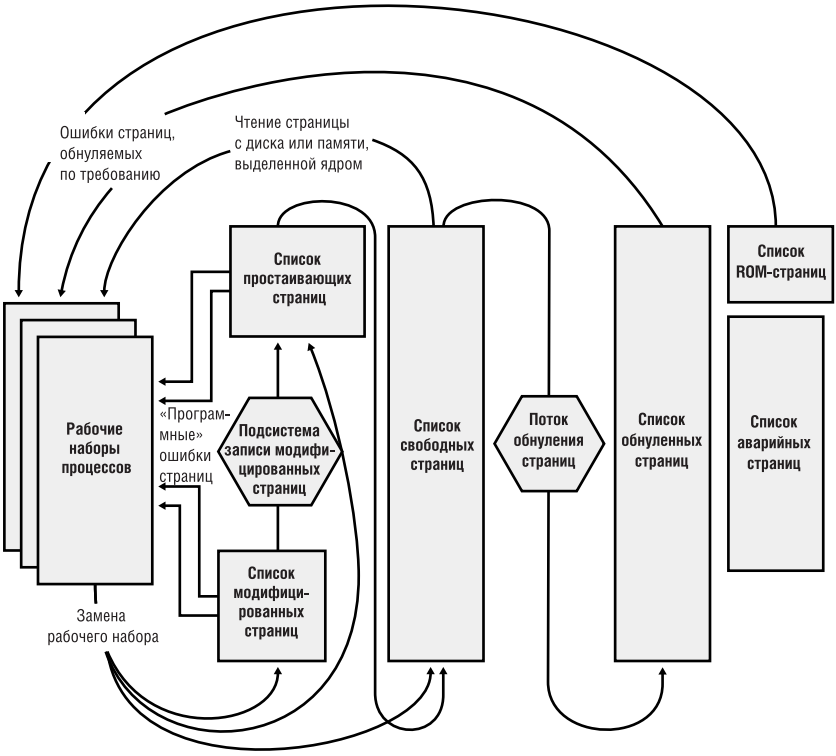


Рис. 7-35. *Схема состояний фреймов страниц*

Одна из причин необходимости обнуления страниц — соответствие требованиям защиты уровня C2: процессам пользовательского режима должны передаваться фреймы обнуленных страниц, чтобы не допустить чтения содержимого памяти предыдущих процессов. Поэтому диспетчер памяти передает процессам пользовательского режима фреймы обнулен-

ных страниц, если только страница не считывается из проецируемого файла. В последнем случае диспетчер памяти использует фреймы необнуленных страниц, инициализируя их данными с диска.

Список обнуленных страниц заполняется страницами из списка свободных страниц системным потоком *обнуления страниц* (zero page thread) — это поток 0 процесса System. Он ждет на объекте-событии, который переходит в свободное состояние при наличии в списке свободных страниц восьми и более страниц. Однако этот поток выполняется, только если не работают другие потоки, поскольку он имеет нулевой приоритет, а наименьший приоритет пользовательского потока — 1.

ПРИМЕЧАНИЕ В Windows Server 2003 и более новых версиях, когда возникает необходимость в обнулении памяти из-за выделения физической страницы драйвером, вызвавшим *MmAllocatePagesForMdl*, или Windows-приложением, вызвавшим *AllocateUserPhysicalPages*, либо когда приложение выделяет большие страницы, диспетчер памяти обнуляет память через более эффективную функцию *MiZeroInParallel*. Она проецирует регионы большего размера, чем поток обнуления страниц, который выполняет свою операцию только над одной страницей единовременно. Кроме того, в многопроцессорных системах эта функция создает дополнительные системные потоки для параллельного выполнения операций обнуления (с поддержкой специфических оптимизаций на NUMA-платформах).

- Если диспетчеру памяти не нужны обнуленные станицы, он сначала обращается к списку свободных страниц и, если тот пуст, переходит к списку простаивающих страниц. Прежде чем диспетчер сможет воспользоваться фреймом страниц из списка простаивающих страниц, он должен проследить ссылку из недействительного PTE (или прототипного PTE), который еще ссылается на этот фрейм, и удалить ее. Поскольку элементы (записи) базы данных PFN содержат обратные указатели на таблицу страниц предыдущего пользователя (или на прототипный PTE, если страницы разделяемые), диспетчер памяти может быстро найти PTE и внести требуемые изменения.
- Когда процессу приходится отдать страницу из своего рабочего набора (из-за ссылки на новую страницу при заполненном рабочем наборе или из-за усечения рабочего набора, инициированного диспетчером памяти), она переходит в список простаивающих страниц (если ее данные не изменялись) или в список модифицированных (если ее данные изменялись, пока она находилась в памяти). По завершении процесса все его закрытые страницы переходят в список свободных страниц. И еще: как только закрывается последняя ссылка на раздел, поддерживаемый страничным файлом, все его страницы тоже переходят в список свободных страниц.

ЭКСПЕРИМЕНТ: наблюдаем за ошибками страниц

Утилита Pfmon из ресурсов Windows 2000 и 2003, а также из Windows XP Support Tools позволяет наблюдать за ошибками страниц по мере их возникновения. Ниже показан фрагмент вывода Pfmon при запуске Notepad и его последующем закрытии. Слово SOFT означает, что ошибка страницы была устранена с помощью одного из переходных списков, а слово HARD — что ошибка страницы потребовала чтения с диска. Обратите внимание на итоговые сведения о подкачке страниц в конце листинга.

```
C:\> pfmon notepad
SOFT: KiUserApcDispatcher : KiUserApcDispatcher
SOFT: LdrInitializeThunk : LdrInitializeThunk
SOFT: 0x77f61016 :  : 0x77f61016
SOFT: 0x77f6105b :  : fltused+0xe00
HARD: 0x77f6105b :  : fltused+0xe00
SOFT: LdrQueryImageFileExecutionOptions :
                        LdrQueryImageFileExecutionOptions
SOFT: RtlAppendUnicodeToString : RtlAppendUnicodeToString
SOFT: RtlInitUnicodeString : RtlInitUnicodeString
...
  notepad Caused   8 faults had   9 Soft  5 Hard faulted VA's
    ntdll Caused  94 faults had  42 Soft  8 Hard faulted VA's
 comdlg32 Caused   3 faults had   0 Soft  3 Hard faulted VA's
  shlwapi Caused   2 faults had   2 Soft  2 Hard faulted VA's
    gdi32 Caused  18 faults had  10 Soft  2 Hard faulted VA's
 kernel32 Caused  48 faults had  36 Soft  3 Hard faulted VA's
   user32 Caused  38 faults had  26 Soft  6 Hard faulted VA's
 advapi32 Caused   7 faults had   6 Soft  3 Hard faulted VA's
   rpcrt4 Caused   6 faults had   4 Soft  2 Hard faulted VA's
 comctl32 Caused   6 faults had   5 Soft  2 Hard faulted VA's
  shell32 Caused   6 faults had   5 Soft  2 Hard faulted VA's
          Caused  10 faults had   9 Soft  5 Hard faulted VA's
 winspool Caused   4 faults had   2 Soft  2 Hard faulted VA's

 PFMON: Total Faults 250
       (KM 74 UM 250 Soft 204, Hard 46, Code 121, Data 129)
```

Подсистема записи модифицированных страниц

При чрезмерном увеличении списка модифицированных страниц либо при уменьшении размера списков обнуленных или простаивающих страниц ниже определенного порогового значения, вычисляемого в ходе загрузки системы и хранящегося в переменной ядра *MmMinimumFreePages*, пробуждается один из двух системных потоков, который записывает страницы на диск и переводит их в список простаивающих. Один из системных потоков (*MiModifiedPageWriter*) записывает модифицированные страницы в странич-

ный файл, а другой (*MiMappedPageWriter*) — в проецируемые файлы. Два потока нужны для того, чтобы избежать тупиковой ситуации, возможной при ошибке страницы в момент записи страниц проецируемых файлов. Эта ошибка потребовала бы свободной страницы в отсутствие таковых, что в свою очередь потребовало бы от подсистемы записи модифицированных страниц создания новых свободных страниц. Поскольку операции ввода-вывода с проецируемыми файлами выполняет второй поток этой подсистемы, он может переходить в состояние ожидания, не блокируя обычные операции ввода-вывода со страничным файлом.

Оба потока выполняются с приоритетом 17, и после инициализации каждый из них ждет на своем объекте-событии. Этот объект переходит в свободное состояние по одной из двух причин.

- Число модифицированных страниц превышает максимум, рассчитанный при инициализации системы (*MmModifiedPageMaximum*) (в настоящее время — 800 страниц для всех систем).
- Число доступных страниц (*MmAvailablePages*) меньше значения *MmMinimumFreePages*.

Подсистема записи модифицированных страниц ждет еще на одном событии (*MiMappedPagesTooOldEvent*), которое устанавливается по истечении предопределенного числа секунд (*MmModifiedPageLifeInSeconds*), указывая, что проецируемые (а не просто измененные) страницы должны быть записаны на диск. (По умолчанию этот интервал равен 300 секундам. Вы можете изменить его, добавив в раздел реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management параметр ModifiedPageLife типа DWORD.) Дополнительное событие используется для того, чтобы снизить риск потери данных при крахе системы или отказе электропитания путем сохранения модифицированных проецируемых страниц, даже если размер списка модифицированных страниц не достиг порогового значения в 800 страниц.

При активизации подсистема записи модифицированных страниц пытается записать на диск как можно больше страниц одним пакетом. Для этого она анализирует поле исходного содержимого PTE в элементах базы данных PFN, которые относятся к страницам, входящим в список модифицированных страниц, и ищет страницы, хранящиеся в непрерывных областях на диске. Подобрав пакет страниц для записи, она выдает запрос на ввод-вывод и после успешного завершения операции ввода-вывода перемещает их в конец списка простаивающих страниц.

При записи страницы к ней может обратиться другой поток. В этом случае счетчики ссылок и числа пользователей в элементе PFN, который представляет физическую страницу, увеличиваются на 1. По окончании операции ввода-вывода подсистема записи модифицированных страниц обнаружит, что счетчик числа пользователей больше не равен 0, и не станет перемещать эту страницу в список простаивающих страниц.

Структуры данных PFN

Хотя записи базы данных PFN имеют фиксированную длину, они могут находиться в нескольких состояниях в зависимости от состояния страницы. Таким образом, отдельные поля могут иметь разный смысл. Состояния записи базы данных PFN иллюстрирует рис. 7-36.

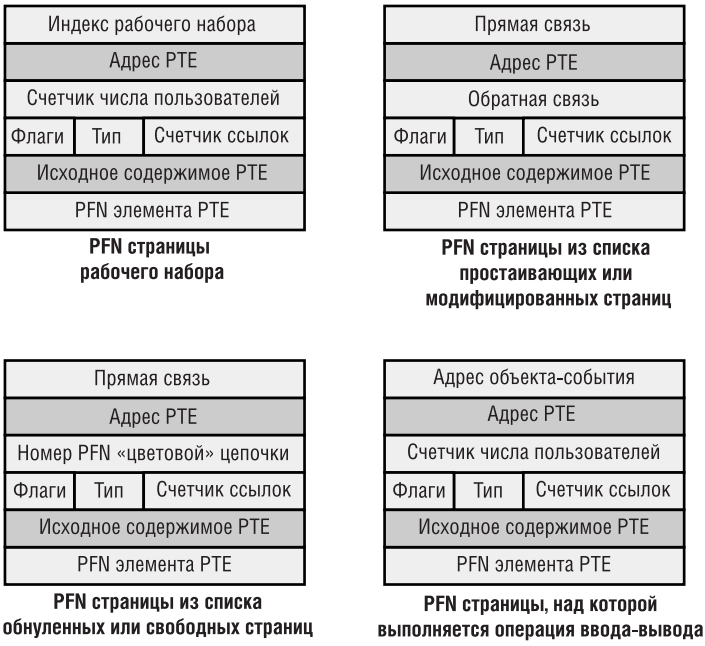


Рис. 7-36. *Состояния записи базы данных PFN*

Некоторые поля одинаковы для нескольких типов PFN, другие специфичны для конкретного типа PFN. Следующие поля встречаются в PFN нескольких типов.

- **Адрес PTE** Виртуальный адрес PTE, указывающего на данную страницу.
- **Счетчик ссылок** Число ссылок на данную страницу. Этот счетчик увеличивается на 1, когда страница впервые добавляется в рабочий набор и/или когда она блокируется в памяти для операции ввода-вывода (например драйвером устройства). Счетчик ссылок уменьшается на 1, когда обнуляется счетчик числа пользователей страницы или когда снимается блокировка страницы в памяти. Когда счетчик числа пользователей становится равным 0, страница больше не принадлежит рабочему набору. Далее, если счетчик ссылок тоже равен 0, страница перемещается в список свободных, простаивающих или модифицированных страниц, и запись базы данных PFN, описывающая эту страницу, соответственно обновляется.
- **Тип** Тип страницы, представленной этим PFN (активная/действительная, переходная, простаивающая, модифицированная, модифицирован-

ная, но не записываемая, свободная, обнуленная, только для чтения или аварийная).

- **Флаги** Информация, содержащаяся в поле флагов, поясняется в таблице 7-21.
- **Исходное содержимое PTE** Все записи базы данных PFN включают исходное содержимое PTE, указывающего на страницу (который может быть прототипным PTE). Сохранение исходного содержимого PTE позволяет восстанавливать его, когда физическая страница больше не резидентна.
- **PFN элемента PTE** Номер физической страницы для виртуальной страницы с таблицей страниц, включающей PTE страницы, к которой относится данный PFN.

Таблица 7-21. *Флаги в записях базы данных PFN*

Флаг	Описание
Состояние модификации	Указывает, модифицирована ли страница (если да, то перед удалением из памяти ее содержимое должно быть сохранено на диске)
Прототипный PTE	Указывает, что PTE, на который ссылается запись PFN, является прототипным. (Например, эта страница доступна для совместного использования.)
Ошибка четности	Указывает, что на физической странице обнаружена ошибка четности или ECC
В процессе чтения	Указывает, что страница загружается в память. Первое поле типа DWORD содержит адрес объекта «событие», который освободится по окончании операции ввода-вывода; также указывает первый PFN при выделении памяти из неподкачиваемого пула
В процессе записи	Указывает, что страница записывается. Первое поле типа DWORD содержит адрес объекта «событие», который освободится по окончании операции ввода-вывода; также указывает последний PFN при выделении памяти из неподкачиваемого пула
Начало пула неподкачиваемой памяти	Для страниц пула неподкачиваемой памяти указывает, что данный PFN является первым для данной области, выделенной в пуле неподкачиваемой памяти
Конец пула неподкачиваемой памяти	Для страниц пула неподкачиваемой памяти указывает, что данный PFN является последним для данной области, выделенной в пуле неподкачиваемой памяти
Ошибка загрузки страницы	Указывает, что при загрузке данной страницы произошла ошибка (в этом случае первое поле PFN содержит код ошибки)

Остальные поля специфичны для PFN конкретных типов. Так, первый PFN на рис. 7-36 представляет активную страницу, входящую в рабочий набор. Поле счетчика числа пользователей (share count) сообщает количество PTE, ссылающихся на данную страницу. (Страницы с атрибутом «только для чтения», «копирование при записи» или «разделяемая, для чтения и записи» могут использоваться сразу несколькими процессами.) В случае страниц с

таблицами страниц это поле содержит количество действительных PTE в таблице страниц. Пока счетчик числа пользователей страницы больше 0, она не удаляется из памяти.

Поле индекса рабочего набора — это индекс в списке рабочего набора (процесса, системы или сеанса), включающем виртуальный адрес, по которому проецируется данная физическая страница. Если страница не входит ни в один рабочий набор, это поле равно 0. Если страница является закрытой, индекс рабочего набора ссылается непосредственно на элемент списка рабочего набора, поскольку страница проецируется только по одному виртуальному адресу. В случае разделяемой страницы индекс рабочего набора представляет собой ссылку, корректность которой гарантируется лишь для первого процесса, объявившего эту страницу действительной. (Остальные процессы будут пытаться по возможности использовать тот же индекс.) Процесс, первым настроивший это поле, обязательно получит корректную ссылку, и ему не надо добавлять в дерево хэшей своего рабочего набора элемент хэша списка рабочего набора, на который указывает виртуальный адрес. Это позволяет уменьшить размер дерева хэшей рабочего набора и ускорить поиск его элементов.

Второй PFN на рис. 7-36 соответствует странице из списка простаивающих или модифицированных страниц. В этом случае элементы списка связаны полями прямых и обратных связей. Эти связи позволяют легко манипулировать страницами при обработке ошибок страниц. Когда страница находится в одном из списков, ее счетчик числа пользователей по определению равен 0 (поскольку она не используется ни в одном рабочем наборе) и поэтому может быть перекрыта обратной связью. Счетчик ссылок также равен 0, если страница находится в одном из списков. Если же он отличен от 0 (из-за выполнения над данной страницей операции ввода-вывода, например записи на диск), страница сначала удаляется из списка.

Третий PFN на рис. 7-36 соответствует странице из списка свободных или обнуленных страниц. Эти записи базы данных PFN связывают не только страницы внутри двух списков, но и — с помощью дополнительного поля — физические страницы «по цвету», т. е. по их местонахождению в кэш-памяти процессора. Windows пытается свести к минимуму лишнюю нагрузку на кэш-память процессора из-за наличия в ней разных физических страниц. Эта оптимизация реализуется за счет того, что Windows по возможности избегает использования одного и того же элемента кэша для двух разных страниц. Для систем с прямым проецированием кэшей оптимальное использование аппаратных возможностей дает существенный выигрыш в производительности.

Четвертый PFN на рис. 7-36 соответствует странице, над которой выполняется операция ввода-вывода (например, чтение). Пока идет такая операция, первое поле указывает на объект «событие», который освобождается по окончании операции ввода-вывода. Если при этом произойдет ошибка, в данное поле будет записан код статуса ошибки Windows, представляющий ошибку ввода-вывода. PFN этого типа используются в разрешении конфликтов ошибок страницы.

ЭКСПЕРИМЕНТ: просмотр записей PFN

Отдельные записи PFN можно исследовать с помощью команды *!pfn* отладчика ядра, указывая PFN как аргумент. (Так, *!pfn 0* сообщает информацию о первой записи, *!pfn 1* — о второй и т. д.) В следующем примере показывается PTE для виртуального адреса 0x50000, PFN страницы с каталогом страниц и сама страница.

```
kd> !pte 50000
00050000  - PDE at C0300000        PTE at C0000140
          contains 00700067      contains 00DAA047
        pfn 00700 --DA--UWV    pfn 00DAA --D--UWV
kd> !pfn 700
  PFN 00000700 at address 827CD800
  flink  00000004 blink / share count 00000010 pteaddress C0300000
  reference count 0001                                color 0
  restore pte 00000080  containing page        00030  Active     ...
  Modified

kd> !pfn daa
  PFN 00000DAA at address 827D77F0
  flink  00000077 blink / share count 00000001 pteaddress C0000140
  reference count 0001                                color 0
  restore pte 00000080  containing page        00700  Active     ...
  Modified
```

Общее состояние физической памяти описывается не только базой данных PFN, но и системными переменными, перечисленными в таблице 7-22.

Таблица 7-22. *Системные переменные, описывающие состояние физической памяти*

Переменная	Описание
MmNumberOfPhysicalPages	Общее количество физических страниц, доступное в системе
MmAvailablePages	Общее количество страниц, доступное в системе: суммарное число страниц в списках обнуленных, свободных и простаивающих страниц
MmResidentAvailablePages	Общее количество физических страниц, которое могло бы быть доступно при минимальном размере всех рабочих наборов процессов

Уведомление о малом или большом объеме памяти

Windows XP и Windows Server 2003 позволяют процессам пользовательского режима получать уведомления, когда памяти мало и/или много. На основе этой информации можно определять характер использования памяти. Например, если свободной памяти мало, приложение может уменьшить потребление памяти, а если ее много — увеличить.

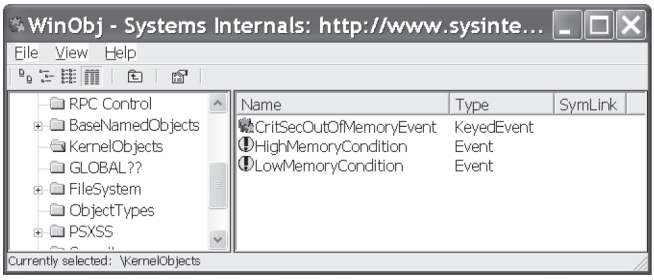
Для уведомления о малом или большом объеме памяти вызовите функцию *CreateMemoryResourceNotification*, указав, какое именно уведомление вас интересует. Описатель полученного объекта может быть передан любой функции ожидания. Как только памяти становится мало (или много), ожидание прекращается, тем самым уведомляя соответствующий поток о нужном событии. В качестве альтернативы можно использовать *QueryMemoryResourceNotification* для запроса о состоянии системной памяти в любой момент.

Уведомление реализуется диспетчером памяти, который переводит в свободное состояние глобально именованный объект «событие» *LowMemoryCondition* или *HighMemoryCondition*. Эти именованные объекты находятся не в обычном каталоге \BaseNamedObjects диспетчера объектов, а в специальном каталоге \KernelObjects. Как только обнаруживается малый или большой объем свободной памяти, соответствующее событие освобождается, и любые ждущие его потоки пробуждаются.

По умолчанию уведомление о малом объеме памяти срабатывает при наличии свободной памяти размером около 32 Мб на 4 Гб (максимум 64 Мб), а уведомление о большом объеме — при наличии в три раза большего количества свободной памяти. Эти значения (в мегабайтах) можно переопределить, добавив DWORD-параметр LowMemoryThreshold или HighMemoryThreshold в раздел реестра HKEY_LOCAL_MACHINE\System\CurrentControlSet\Session Manager\Memory Management.

ЭКСПЕРИМЕНТ: просмотр событий уведомления ресурса памяти

Для просмотра событий уведомления ресурса памяти (memory resource notification events) запустите Winobj (*www.sysinternals.com*) и щелкните каталог KernelObjects. В правой секции окна вы увидите события *LowMemoryCondition* и *HighMemoryCondition*.



Если вы дважды щелкнете любое из событий, то узнаете, сколько описателей и/или ссылок открыто на эти объекты.

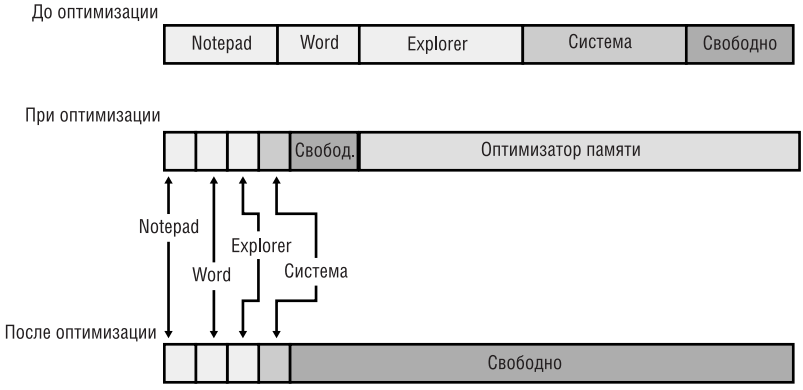
Чтобы выяснить, есть ли в системе процессы, запросившие уведомления о ресурсе памяти, ищите в таблице описателей ссылки на «LowMemoryCondition» или «HighMemoryCondition». Это можно сделать в Process Explorer (команда Handle в меню Find) или в утилите Oh.exe из ресурсов Windows. (О том, что такое таблица описателей, см. раздел «Диспетчер объектов» главы 3.)

Оптимизаторы памяти — миф или реальность?

При серфинге по Web вы наверняка нередко видели всплывающие окна в браузере с рекламой наподобие «Дефрагментируйте память и повысьте производительность» или «Избавьтесь от сбоев приложений и системы и освободите неиспользуемую память». Такие ссылки обычно ведут к утилитам, авторы которых обещают сделать все и даже больше. А работают ли они на самом деле?

Оптимизаторы памяти обычно предоставляют UI, где выводятся график под названием «доступная память» и линия, отражающая нижнее пороговое значение, начиная с которого утилита вступает в действие. Еще одна линия, как правило, показывает объем памяти, который оптимизатор попытается освободить. Вы можете настроить один или оба уровня, а также запускать оптимизацию вручную или по расписанию. Некоторые утилиты также показывают список процессов, выполняемых в системе. Когда начинается оптимизация, счетчик доступной памяти в утилите увеличивается, иногда весьма резко, сообщая тем самым, что утилита действительно освобождает память для ваших приложений. Но на самом деле подобные утилиты просто вызывают обнуление полезной памяти, искусственно увеличивая объем свободной памяти.

Оптимизаторы памяти выделяют, а потом освобождают большие объемы виртуальной памяти. На иллюстрации ниже показано, какое влияние оказывают оптимизаторы памяти на систему.



Полоска «до оптимизации» отражает рабочие наборы и свободную память до оптимизации. На полоске «при оптимизации» видно, что оптимизатор создает высокую потребность в памяти, вызывая массу ошибок страниц в течение короткого времени. В ответ диспетчер памяти увеличивает рабочий набор оптимизатора памяти. Это расширение рабочего набора происходит за счет свободной памяти, а когда свободная память заканчивается, то и за счет рабочих наборов других процессов. Полоска «после оптимизации» демонстрирует, что после

см. след. стр.

освобождения своей памяти оптимизатором диспетчер памяти переводит все страницы, которые были закреплены за оптимизатором, в список свободных страниц. Там они в конечном счете заполняются нулями потоком обнуления страниц, а затем перемещаются в список обнуленных страниц, что и дает вклад в увеличение счетчика доступной памяти. Большинство оптимизаторов скрывают резкое уменьшение свободной памяти на первом этапе, но, запустив диспетчер задач при оптимизации, вы легко заметите, что такое падение объема свободной памяти действительно имеет место.

Хотя получение большего объема свободной памяти может показаться полезным, это не так. Когда оптимизаторы вызывают подъем значений счетчика доступной памяти, они заставляют систему выгружать из памяти код и данные других процессов. Если, например, вы работаете с Word, то текст открытых документов и код этой программы до оптимизации являются частью рабочего набора Word (и, следовательно, находятся в физической памяти), а после оптимизации придется вновь считывать их с диска, как только вы захотите продолжить работу с документами. На серверах падение производительности бывает просто колоссальным, так как на них после оптимизации могут быть отброшены файловые данные, которые кэшировались в списке простаивающих страниц и системном рабочем наборе (то же самое относится к коду и данным, используемым любыми выполняемыми серверными приложениями).

Некоторые разработчики оптимизаторов памяти заявляют, будто их утилиты освобождают память, бессмысленно занимаемую неиспользуемыми процессами, например теми, значки которых помещаются в секцию индикаторов панели задач. Это могло бы быть правдой только в том случае, если бы к моменту оптимизации у этих процессов были рабочие наборы существенного размера. Но Windows автоматически усекает рабочие наборы простаивающих процессов, и поэтому подобные заявления не соответствуют истине. Все, что нужно для реальной оптимизации, делает диспетчер памяти.

Авторы оптимизаторов памяти также утверждают, что их утилиты дефрагментируют память. Действительно, выделение и последующее освобождение большого объема виртуальной памяти может в качестве побочного эффекта привести к появлению больших блоков непрерывной свободной памяти. Однако, так как виртуальная память скрывает реальную структуру физической памяти от процессов, они не могут получить прямой выигрыш от виртуальной памяти, спроецированной на непрерывную область физической памяти. По мере выполнения процессов и периодического расширения-усечения их рабочих наборов физическая память, сопоставленная с занимаемой ими виртуальной памятью, может стать фрагментированной несмотря на наличие больших непрерывных областей. К тому же, любой незначительный выигрыш от дефрагментации свободной физической памяти с лихвой

перекрывается негативным эффектом от выгрузки из памяти используемого кода и данных.

Наконец, можно услышать, что оптимизаторы возвращают память, потерянную в результате утечек. Это, наверное, самое ложное утверждение. Диспетчеру памяти всегда известно, какая физическая и виртуальная память принадлежит тому или иному процессу. Однако, если процесс выделяет память, а потом не освобождает ее из-за какой-то ошибки (это событие и называется утечкой), диспетчер памяти не в состоянии распознать, что выделенная память больше не будет использоваться, и поэтому вынужден ждать завершения процесса, чтобы отобрать эту память. Даже у процессов, которые вызывают утечку памяти и не завершаются, диспетчер памяти в конечном счете, в результате усечения рабочего набора отберет все физические страницы, связанные с утекающей виртуальной памятью. Страницы последней будут отправлены в страничный файл, а физическая память будет освобождена для использования в других целях. Таким образом, утечка памяти лишь ограниченно влияет на доступную физическую память. По-настоящему она влияет на потребление виртуальной памяти, которое хорошо заметно по счетчикам PF Usage и Commit Charge в диспетчере задач. Никакая утилита ничего не сможет сделать с пустым расходом виртуальной памяти, если только не «убьет» процессы, поглощающие эту память.

Короче говоря, если бы от оптимизаторов памяти был хоть какой-нибудь толк, разработчики Microsoft давно интегрировали бы такую технологию в ядро Windows.

Резюме

В этой главе мы изучили, как диспетчер памяти управляет виртуальной памятью. Как и в большинстве других современных операционных систем, в Windows у каждого процесса имеется закрытое адресное пространство, защищенное от доступа других процессов, но обеспечивающее эффективное и безопасное разделение памяти несколькими процессами. Поддерживаются и такие дополнительные возможности, как включение (inclusion) проецируемых файлов и разреженная память. Подсистема окружения Windows предоставляет приложениям большинство функций диспетчера памяти через Windows API.

Диспетчер памяти везде, где это возможно, использует алгоритмы отложенной оценки, что помогает избежать выполнения лишних операций, отнимающих много времени. Такие операции выполняются только по необходимости. Диспетчер памяти также является самонастраивающейся подсистемой, которая автоматически адаптируется для работы как на мощных многопроцессорных серверах, так и на однопроцессорных персональных компьютерах.

В этой главе мы не затронули такой аспект, как тесная интеграция диспетчера памяти с диспетчером кэша, — об этом будет рассказано в главе 11. А теперь давайте перейдем к детальному рассмотрению механизмов защиты Windows .

ГЛАВА 8

Защита

Защита конфиденциальных данных от несанкционированного доступа очень важна в любой среде, где множество пользователей обращается к одним и тем же физическим или сетевым ресурсам. У операционной системы, как и у отдельных пользователей, должна быть возможность защиты файлов, памяти и конфигурационных параметров от нежелательного просмотра и внесения изменений. Безопасность операционной системы обеспечивается такими очевидными механизмами, как учетные записи, пароли и защита файлов. Но она требует и менее очевидных механизмов — защиты операционной системы от повреждения, запрета непривилегированным пользователям определенных действий (например, перезагрузки компьютера), предотвращения неблагоприятного воздействия программ одних пользователей на программы других пользователей или на операционную систему.

В этой главе мы поясним, как жесткие требования к защите повлияли на внутреннее устройство и реализацию Microsoft Windows.

Классы безопасности

Четкие стандарты безопасности программного обеспечения, в том числе операционных систем, помогают правительству, корпорациям и индивидуальным пользователям защищать хранящиеся в компьютерных системах данные, составляющие личную и коммерческую тайну. Текущий стандарт на рейтинги безопасности, применяемый в США и многих других странах, — Common Criteria (CC). Однако, чтобы понять средства защиты, существующие в Windows, нужно знать историю системы рейтингов безопасности, повлиявшей на архитектуру системы защиты Windows, — Trusted Computer System Evaluation Criteria (TCSEC).

Trusted Computer System Evaluation Criteria

Национальный центр компьютерной безопасности (National Computer Security Center, NCSC, *www.radium.ncsc.mil/tpep*) был создан в 1981 году в Агентстве национальной безопасности (NSA) Министерства обороны США. Одна из задач NCSC заключалась в определении рейтингов безопасности (см. таблицу 8-1), отражающих степень защищенности коммерческих операционных систем, сетевых компонентов и приложений. Эти рейтинги, детальное описание которых вы найдете по ссылке *www.radium.ncsc.mil/tpep/library/*

rainbow/5200.28-STD.html, были определены в 1983 году и часто называются «Оранжевой книгой» («Orange Book»).

Таблица 8-1. *Рейтинги безопасности TCSEC*

Рейтинг	Название
A1	Verified Design
B3	Security Domains
B2	Structured Protection
B1	Labeled Security Protection
C2	Controlled Access Protection
C1	Discretionary Access Protection (устарел)
D	Minimal Protection

Стандарт TCSEC состоит из рейтингов «уровней доверия» («levels of trust» ratings), где более высокие уровни строятся на более низких за счет последовательного ужесточения требований к безопасности и проверке. Ни одна операционная система не соответствует уровню A1 (Verified Design). Хотя некоторым операционным системам присвоен один из уровней B, уровень C2 считается достаточным и является высшим для операционных систем общего назначения.

В июле 1995 года Windows NT 3.5 (Workstation и Server) с Service Pack 3 первой из всех версий Windows NT получила подтверждение об уровне безопасности C2. В марте 1999 года организация ITSEC (Information Technology Security) правительства Великобритании присвоила Windows NT 4 с Service Pack 3 уровень E3, эквивалентный американскому уровню C2. Windows NT 4 с Service Pack 6a получила уровень C2 для сетевой и автономной конфигураций.

Какие требования предъявляются к уровню безопасности C2? Основные требования (они остались прежними) перечислены ниже.

- *Механизм безопасной регистрации*, требующий уникальной идентификации пользователей. Доступ к компьютеру предоставляется лишь после аутентификации.
- *Управление избирательным доступом*, позволяющее владельцу ресурса определять круг лиц, имеющих доступ к ресурсу, а также их права на операции с этим ресурсом. Владелец предоставляет пользователям и их группам различные права доступа.
- *Аудит безопасности*, обеспечивающий возможность регистрации событий, связанных с защитой, а также любых попыток создания, удаления и обращения к системным ресурсам. При входе регистрируются идентификационные данные всех пользователей, что позволяет легко выявить любого, кто попытался выполнить несанкционированную операцию.
- *Защита при повторном использовании объектов*, которая предотвращает просмотр одним из пользователей данных, удаленных из памяти другим, а также доступ к памяти, освобожденной предыдущим пользователем. Например, в некоторых операционных системах можно создать но-

вый файл определенной длины, а затем просмотреть те данные, которые остались на диске и попали в область, отведенную под новый файл. Среди этих данных может оказаться конфиденциальная информация, хранившаяся в недавно удаленном файле другого пользователя. Так что защита при повторном использовании объектов устраняет потенциальную дыру в системе безопасности, заново инициализируя перед выделением новому пользователю все объекты, включая файлы и память.
Windows также отвечает двум требованиям защиты уровня B.

- *Функциональность пути доверительных отношений* (trusted path functionality), которая предотвращает перехват имен и паролей пользователей программами — троянскими конями. Эта функциональность реализована в Windows в виде входной сигнальной комбинации клавиш Ctrl+Alt+Del и не может быть перехвачена непривилегированными приложениями. Такая комбинация клавиш, известная как SAS (secure attention sequence), вызывает диалоговое окно входа в систему, обходя вызов его фальшивого эквивалента из троянского коня.
- *Управление доверительными отношениями* (trusted facility management), которое требует поддержки набора ролей (различных типов учетных записей) для разных уровней работы в системе. Например, функции администрирования доступны только по одной группе учетных записей — Administrators (Администраторы).
Windows соответствует всем перечисленным требованиям.

Common Criteria

В январе 1996 года США, Великобритания, Германия, Франция, Канада и Нидерланды опубликовали совместно разработанную спецификацию оценки безопасности Common Criteria for Information Technology Security Evaluation (CCITSE). Эта спецификация, чаще называемая Common Criteria (CC) (*csrc.nist.gov/cc*), является международным стандартом оценки степени защищенности продуктов.

CC гибче уровней доверия TCSEC и по структуре ближе ITSEC, чем TCSEC. CC включает две концепции:

- *профиля защиты* (Protection Profile, PP) — требования к безопасности разбиваются на группы, которые легко определять и сравнивать;
- *объекта защиты* (Security Target, ST) — предоставляет набор требований к защите, которые могут быть подготовлены с помощью PP.

Windows 2000 оценивалась на соответствие требованиям Controlled Access PP, эквивалентным TCSEC C2, и на соответствие дополнительным требованиям Common Criteria в октябре 2002 года. Подробности можно найти по ссылкам *niap.nist.gov/cc-scheme.html* и *niap.nist.gov/cc-scheme/CCEVS-VID402-ST.pdf*. К значимым требованиям, не включенным в Controlled Access PP, но предъявляемым по условиям Windows 2000 Security Target, относятся:

- *функции управления избирательным доступом* (Discretionary Access Control Functions), основанные на применении криптографии. Они реа-

лизуются файловой системой Encrypting File System и Data Protection API в Windows 2000;

- *политика управления избирательным доступом* (Discretionary Access Control Policy) для дополнительных пользовательских объектов данных (User Data Objects), например объекты Desktop и WindowStation (реализуются подсистемой поддержки окон Windows 2000), а также объекты Active Directory (реализуются службой каталогов в Windows 2000);
- *внутренняя репликация* (Internal Replication) для гарантированной синхронизации элементов данных, связанных с защитой, между физически раздельными частями системы Windows 2000 как распределенной операционной системы. Это требование реализуется службой каталогов Windows 2000 с применением модели репликации каталогов с несколькими хозяевами;
- *утилизация ресурсов* (Resource Utilization) для физических пространств дисков. Это требование реализуется файловой системой NTFS в Windows 2000;
- *блокировка интерактивного сеанса* (Interactive Session Locking) и *путь доверительных отношений* (Trusted Path) для первоначального входа пользователя (initial user logging on). Это требование реализуется компонентом Winlogon в Windows 2000;
- *защита внутренней передачи данных* (Internal Data Transfer Protection), чтобы обезопасить данные от раскрытия и несанкционированной модификации при передаче между физически раздельными частями системы Windows 2000 как распределенной операционной системы. Это требование реализуется службой IPSEC в Windows 2000;
- *систематическое устранение обнаруживаемых недостатков в системе защиты* (Systematic Flaw Remediation). Это требование реализуется Microsoft Security Response Center и Windows Sustained Engineering Team.

На момент написания этой книги Windows XP Embedded, Windows XP Professional и Windows Server 2003 все еще проходили оценку на соответствие Common Criteria. Набор критериев расширен по сравнению с тем, который применялся к Windows 2000. Комитет Common Criteria в настоящее время рассматривает Windows XP и Windows Server 2003 (Standard, Enterprise и Datacenter Edition) для оценки технологий следующих типов (см. *niap.nist.gov/cc-scheme/in_evaluation.html*):

- распределенной операционной системы;
- защиты конфиденциальных данных;
- управления сетью;
- службы каталогов;
- брандмауэра;
- VPN (Virtual Private Network);
- управления рабочим столом;

- инфраструктуры открытого ключа (Public Key Infrastructure, PKI);
- выдачи и управления сертификатами открытого ключа;
- встраиваемой операционной системы.

Компоненты системы защиты

Ниже перечислены главные компоненты и базы данных, на основе которых реализуется защита в Windows.

- **Монитор состояния защиты (Security Reference Monitor, SRM)** Компонент исполнительной системы (\Windows\System32\ Ntoskrnl.exe), отвечающий за определение структуры данных маркера доступа для представления контекста защиты, за проверку прав доступа к объектам, манипулирование привилегиями (правами пользователей) и генерацию сообщений аудита безопасности.
- **Подсистема локальной аутентификации (local security authentication subsystem, LSASS)** Процесс пользовательского режима, выполняющий образ \Windows\System32\Lsass.exe, который отвечает за политику безопасности в локальной системе (например, круг пользователей, имеющих право на вход в систему, правила, связанные с паролями, привилегии, выдаваемые пользователям и их группам, параметры аудита безопасности системы), а также за аутентификацию пользователей и передачу сообщений аудита безопасности в Event Log. Основную часть этой функциональности реализует сервис локальной аутентификации Lsasrv (\Windows\System32\Lsasrv.dll) — DLL-модуль, загружаемый Lsass.
- **База данных политики LSASS** База данных, содержащая параметры политики безопасности локальной системы. Она хранится в разделе реестра HKLM\SECURITY и включает следующую информацию: каким доменам доверена аутентификация попыток входа в систему, кто имеет права на доступ к системе и каким образом, кому предоставлены те или иные привилегии и какие виды аудита следует выполнять. База данных политики LSASS также хранит «секреты», которые включают в себя регистрационные данные, применяемые для входа в домены и при вызове Windows-сервисов (о Windows-сервисах см. главу 5).
- **Диспетчер учетных записей безопасности (Security Accounts Manager, SAM)** Набор подпрограмм, отвечающих за поддержку базы данных, которая содержит имена пользователей и группы, определенные на локальной машине. Служба SAM, реализованная как \Windows\System32\ Samsrv.dll, выполняется в процессе Lsass.
- **База данных SAM** База данных, которая в системах, отличных от контроллеров домена, содержит информацию о локальных пользователях и группах вместе с их паролями и другими атрибутами. На контроллерах домена SAM содержит определение и пароль учетной записи администратора, имеющего права на восстановление системы. Эта база данных хранится в разделе реестра HKLM\SAM.

- **Active Directory** Служба каталогов, содержащая базу данных со сведениями об объектах в домене. *Домен* — это совокупность компьютеров и сопоставленных с ними групп безопасности, которые управляются как единое целое. Active Directory хранит информацию об объектах домена, в том числе о пользователях, группах и компьютерах. Сведения о паролях и привилегиях пользователей домена и их групп содержатся в Active Directory и реплицируются на компьютеры, выполняющие роль контроллеров домена. Сервер Active Directory, реализованный как \Windows\System32\Ntdsa.dll, выполняется в процессе Lsass. Подробнее об Active Directory см. главу 13.
- **Пакеты аутентификации** DLL-модули, выполняемые в контексте процесса Lsass и клиентских процессов и реализующие политику аутентификации в Windows. DLL аутентификации отвечает за проверку пароля и имени пользователя, а также за возврат LSASS (в случае успешной проверки) детальной информации о правах пользователя, на основе которой LSASS генерирует маркер (token).
- **Процесс входа (Winlogon)** Процесс пользовательского режима (\Windows\System32\Winlogon.exe), отвечающий за поддержку SAS и управление сеансами интерактивного входа в систему. Например, при регистрации пользователя Winlogon создает оболочку — пользовательский интерфейс.
- **GINA (Graphical Identification and Authentication)** DLL пользовательского режима, выполняемая в процессе Winlogon и применяемая для получения пароля и имени пользователя или PIN-кода смарт-карты. Стандартная GINA хранится в \Windows\System32\Msgina.dll.
- **Служба сетевого входа (Netlogon)** Windows-сервис (\Windows\System32 \Netlogon.dll), устанавливающий защищенный канал с контроллером домена, по которому посылаются запросы, связанные с защитой, например для интерактивного входа (если контроллер домена работает под управлением Windows NT), или запросы на аутентификацию от LAN Manager либо NT LAN Manager (v1 и v2).
- **Kernel Security Device Driver (KSecDD)** Библиотека функций режима ядра, реализующая интерфейсы LPC (local procedure call), которые используются другими компонентами защиты режима ядра — в том числе шифрующей файловой системой (Encrypting File System, EFS) — для взаимодействия с LSASS в пользовательском режиме. KsecDD содержится в \Windows\System32\Drivers\Ksecdd.sys.

На рис. 8-1 показаны взаимосвязи между некоторыми из этих компонентов и базами данных, которыми они управляют.

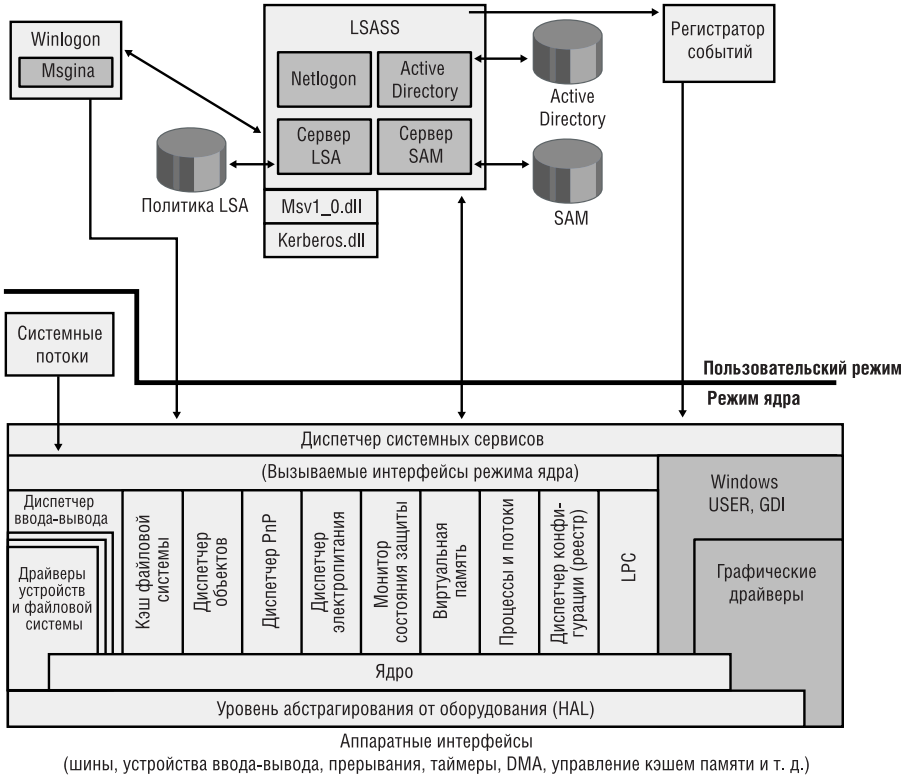
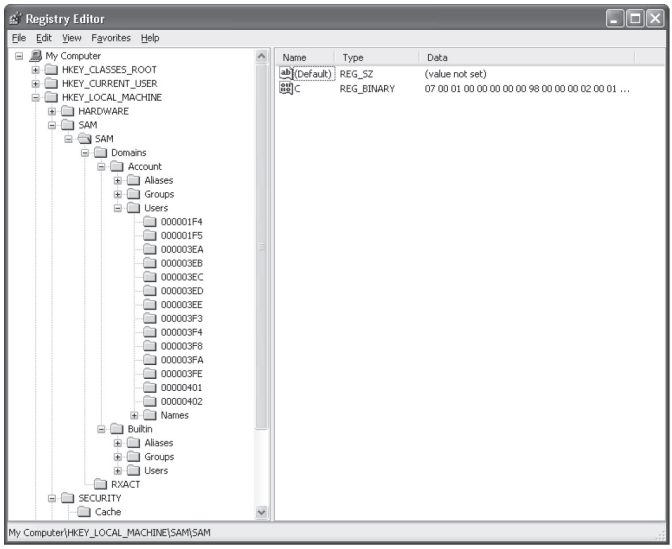


Рис. 8-1. *Компоненты защиты Windows*

ЭКСПЕРИМЕНТ: просмотр содержимого HKLM\SAM и HKLM\Security

Дескрипторы защиты, сопоставленные с разделами реестра SAM и Security, блокируют доступ по любой учетной записи, кроме Local System. Один из способов получить доступ к этим разделам — сбросить их защиту, но это может ослабить безопасность системы. Другой способ — запустить Regedit.exe под учетной записью Local System; такой способ поддерживается утилитой PsExec (*www.sysinternals.com*), которая позволяет запускать процессы под этой учетной записью:

```
C:>psexec -s -i -d c:\windows\regedit.exe
```



SRM, выполняемый в режиме ядра, и LSASS, работающий в пользовательском режиме, взаимодействуют по механизму LPC (см. главу 3). При инициализации системы SRM создает порт SeRmCommandPort, к которому подключается LSASS. Процесс Lsass при запуске создает LPC-порт SeLsaCommandPort. К этому порту подключается SRM. В результате формируются закрытые коммуникационные порты. SRM создает раздел общей памяти для передачи сообщений длиннее 256 байтов и передает его описатель при запросе на соединение. После соединения SRM и LSASS на этапе инициализации системы они больше не прослушивают свои порты. Поэтому никакой пользовательский процесс не сможет подключиться к одному из этих портов.

Рис. 8-2 иллюстрирует коммуникационные пути после инициализации системы.

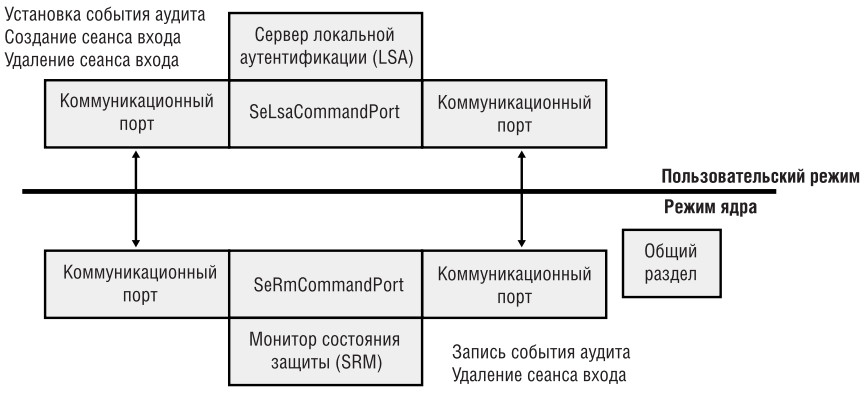


Рис. 8-2. *Взаимодействие между SRM и LSASS*

Защита объектов

Защита объектов и протоколирование обращений к ним — вот сущность управления избирательным доступом и аудита. Защищаемые объекты Windows включают файлы, устройства, почтовые ящики, каналы (именованные и анонимные), задания, процессы, потоки, события, пары событий, мьютексы, семафоры, порты завершения ввода-вывода, разделы общей памяти, LPC-порты, ожидаемые таймеры, маркеры доступа, тома, объекты WindowStation, рабочие столы, сетевые ресурсы, сервисы, разделы реестра, принтеры и объекты Active Directory.

Поскольку системные ресурсы, экспортируемые в пользовательский режим (и поэтому требующие проверки защиты), реализуются как объекты режима ядра, диспетчер объектов играет ключевую роль в их защите (о диспетчере объектов см. главу 3). Для контроля за операциями над объектом система защиты должна быть уверена в правильности идентификации каждого пользователя. Именно по этой причине Windows требует от пользователя входа с аутентификацией, прежде чем ему будет разрешено обращаться к системным ресурсам. Когда какой-либо процесс запрашивает описатель объекта, диспетчер объектов и система защиты на основе идентификационных данных вызывающего процесса определяют, можно ли предоставить ему описатель, разрешающий доступ к нужному объекту.

Контекст защиты потока может отличаться от контекста защиты его процесса. Этот механизм называется *олицетворением* (impersonation), или подменой. При олицетворении механизмы проверки защиты используют вместо контекста защиты процесса контекст защиты потока, а без олицетворения — контекст защиты процесса, которому принадлежит поток. Важно не забывать, что все потоки процесса используют одну и ту же таблицу описателей, поэтому, когда поток открывает какой-нибудь объект (даже при олицетворении), все потоки процесса получают доступ к этому объекту.

Проверка прав доступа

Модель защиты Windows требует, чтобы поток заранее — еще до открытия объекта — указывал, какие операции он собирается выполнять над этим объектом. Система проверяет тип доступа, запрошенный потоком, и, если такой доступ ему разрешен, он получает описатель, позволяющий ему (и другим потокам того же процесса) выполнять операции над объектом. Как уже говорилось в главе 3, диспетчер объектов регистрирует права доступа, предоставленные для данного описателя, в таблице описателей, принадлежащей процессу.

Одно из событий, заставляющее диспетчер объектов проверять права доступа, — открытие процессом существующего объекта по имени. При открытии объекта по имени диспетчер объектов ищет его в своем пространстве имен. Если этого объекта нет во вторичном пространстве имен (например, в пространстве имен реестра, принадлежащем диспетчеру конфигура-

ции, или в пространстве имен файловой системы, принадлежащем драйверу файловой системы), диспетчер объектов вызывает внутреннюю функцию *ObpCreateHandle*. Как и подсказывает ее имя, она создает элемент в таблице описателей, который сопоставляется с объектом. Однако *ObpCreateHandle* вызывает функцию исполнительной системы *ExCreateHandle* и создает описатель, только если другая функция диспетчера объектов, *ObpIncrementHandleCount*, сообщает, что поток имеет право на доступ к данному объекту. Правда, реальную проверку прав доступа осуществляет другая функция диспетчера объектов, *ObCheckObjectAccess*, которая возвращает результаты проверки функции *ObpIncrementHandleCount*.

ObpIncrementHandleCount передает *ObCheckObjectAccess* удостоверения защиты потока, открывающего объект, типы запрошенного им доступа (чтение, запись, удаление и т. д.), а также указатель на объект. *ObCheckObjectAccess* сначала блокирует защиту объекта и контекст защиты потока. Блокировка защиты объекта предотвращает ее изменение другим потоком в процессе проверки прав доступа, а блокировка контекста защиты потока не дает другому потоку того же или другого процесса изменить идентификационные данные защиты первого потока при проверке его прав доступа. Далее *ObCheckObjectAccess* вызывает метод защиты объекта, чтобы получить параметры защиты объекта (описание методов объектов см. в главе 3). Вызов метода защиты может привести к вызову функции из другого компонента исполнительной системы, но многие объекты исполнительной системы полагаются на стандартную поддержку управления защитой, предлагаемую системой.

Если компонент исполнительной системы, определяя объект, не собирается заменять стандартную политику безопасности, он помечает тип этих объектов как использующий стандартную защиту. Всякий раз, когда SRM вызывает метод защиты объекта, он сначала проверяет, использует ли объект стандартную защиту. Объект со стандартной защитой хранит информацию о защите в своем заголовке и предоставляет метод защиты с именем *SeDefaultObjectMethod*. Объект, не использующий стандартную защиту, должен сам поддерживать информацию о защите и предоставлять собственный метод защиты. Стандартную защиту используют такие объекты, как мьютексы, события и семафоры. Пример объекта с нестандартной защитой — файл. У диспетчера ввода-вывода, определяющего объекты типа «файл», имеется драйвер файловой системы, который управляет защитой своих файлов (или решает не реализовать ее). Таким образом, когда система запрашивает информацию о защите объекта «файл», представляющего файл на томе NTFS, она получает эту информацию от драйвера файловой системы NTFS, который в свою очередь получает ее от метода защиты объекта «файл», принадлежащего диспетчеру ввода-вывода. Заметьте, что при открытии файла *ObCheckObjectAccess* не выполняется, так как объекты «файл» находятся во вторичных пространствах имен; система вызывает метод защиты объекта «файл», только если поток явно запрашивает или устанавливает параметры защиты файла (например, через Windows-функции *SetFileSecurity* или *GetFileSecurity*).

Получив информацию о защите объекта, *ObCheckObjectAccess* вызывает SRM-функцию *SeAccessCheck*, на которую опирается вся модель защиты Windows. Она принимает параметры защиты объекта, идентификационные данные защиты потока (в том виде, в каком они получены *ObCheckObjectAccess*) и тип доступа, запрашиваемый потоком. *SeAccessCheck* возвращает True или False в зависимости от того, предоставляет ли она потоку запрошенный тип доступа к объекту.

Другое событие, заставляющее диспетчер объектов выполнять проверку прав доступа, — ссылка процесса на объект по существующему описателю. Подобные ссылки часто делаются косвенно, например при манипуляциях с объектом через Windows API с передачей его описателя. Допустим, поток, открывающий файл, запрашивает доступ для чтения из файла. Если у потока есть соответствующие права, определяемые его контекстом защиты и параметрами защиты файла, диспетчер объектов создает описатель данного файла в таблице описателей, которая принадлежит процессу — владельцу этого потока. Информация о предоставленном процессу типе доступа сопоставляется с описателем и сохраняется диспетчером объектов.

Впоследствии поток может попытаться что-то записать в этот файл через Windows-функцию *WriteFile*, передав в качестве параметра описатель файла. Системный сервис *NtWriteFile*, который *WriteFile* вызовет через Ntdll.dll, обратится к функции диспетчера объектов *ObReferenceObjectByHandle*, чтобы получить указатель на объект «файл» по его описателю. *ObReferenceObjectByHandle* принимает запрошенный тип доступа как параметр. Найдя в таблице описателей элемент, соответствующий нужному описателю, *ObReferenceObjectByHandle* сравнит запрошенный тип доступа с тем, который был предоставлен при открытии файла. В данном случае *ObReferenceObjectByHandle* укажет, что операция записи должна завершиться неудачно, так как вызывающий поток, открывая файл, не получил право на его запись.

Функции защиты Windows также позволяют Windows-приложениям определять собственные закрытые объекты и вызывать SRM-сервисы для применения к этим объектам средств защиты Windows. Многие функции режима ядра, используемые диспетчером объектов и другими компонентами исполнительной системы для защиты своих объектов, экспортируются в виде Windows-функций пользовательского режима. Например, эквивалентом *SeAccessCheck* для пользовательского режима является *AccessCheck*. Таким образом, Windows-приложения могут применять модель защиты Windows и интегрироваться с интерфейсами аутентификации и администрирования этой операционной системы.

Сущность модели защиты SRM отражает математическое выражение с тремя входными параметрами: идентификационными данными защиты потока, запрошенным типом доступа и информацией о защите объекта. Его результат — значения «да» или «нет», которые определяют, предоставит ли модель защиты запрошенный тип доступа. В следующих разделах мы пого-

ворим об этих входных параметрах и алгоритме проверки прав доступа в модели защиты.

Идентификаторы защиты

Для идентификации объектов, выполняющих в системе различные действия, Windows использует не имена (которые могут быть не уникальными), а *идентификаторы защиты* (security identifiers, SID). SID имеются у пользователей, локальных и доменных групп, локальных компьютеров, доменов и членов доменов. SID представляет собой числовое значение переменной длины, формируемое из номера версии структуры SID, 48-битного кода агента идентификатора и переменного количества 32-битных кодов субагентов и/или *относительных идентификаторов* (relative identifiers, RID). Код агента идентификатора (identifier authority value) определяет агент, выдавший SID. Таким агентом обычно является локальная система или домен под управлением Windows. Коды субагентов идентифицируют попечителей, уполномоченных агентом, который выдал SID, а RID — не более чем средство создания уникальных SID на основе общего базового SID (common-based SID). Поскольку длина SID довольно велика и Windows старается генерировать истинно случайные значения для каждого SID, вероятность появления двух одинаковых SID практически равна нулю.

В текстовой форме каждый SID начинается с префикса *S*, за которым следуют группы чисел, разделяемые дефисами, например:

```
S-1-5-21-1463437245-1224812800-863842198-1128
```

В этом SID номер версии равен 1, код агента идентификатора — 5 (центр безопасности Windows), далее идут коды четырех субагентов и один RID в конце (1128). Этот SID относится к домену, так что локальный компьютер этого домена получит SID с тем же номером версии и кодом агента идентификатора; кроме того, в нем будет столько же кодов субагентов.

SID назначается компьютеру при установке Windows (программой Windows Setup). Далее Windows назначает SID локальным учетным записям на этом компьютере. SID каждой локальной учетной записи формируется на основе SID компьютера с добавлением RID. RID пользовательской учетной записи начинается с 1000 и увеличивается на 1 для каждого нового пользователя или группы. Аналогичным образом Dcpromo.exe — утилита, применяемая при создании нового домена Windows, — выдает SID только что созданному домену. Новые учетные записи домена получают SID, формируемые на основе SID домена с добавлением RID (который также начинается с 1000 и увеличивается на 1 для каждого нового пользователя или группы). RID с номером 1028 указывает на то, что его SID является 29-м, выданным доменом.

Многим предопределенным учетным записям и группам Windows выдает SID, состоящие из SID компьютера или домена и предопределенного RID. Так, RID учетной записи администратора равен 500, а RID гостевой учетной записи — 501. Например, в основе SID учетной записи локального администратора лежит SID компьютера, к которому добавлен RID, равный 500:

```
S-1-5-21-13124455-12541255-61235125-500
```

Для групп Windows также определяет ряд встроенных локальных и доменных SID. Например, SID, представляющий любую учетную запись, называется Everyone или World и имеет вид S-1-1-0. Еще один пример — сетевая группа, т. е. группа, пользователи которой зарегистрировались на данном компьютере из сети. SID сетевой группы имеет вид S-1-5-2. Список некоторых общеизвестных SID приведен в таблице 8-2 (полный список см. в документации Platform SDK).

Наконец, Winlogon создает уникальный SID для каждого интерактивного сеанса входа. SID входа, как правило, используется в элементе списка управления доступом (access-control entry, ACE), который разрешает доступ на время сеанса входа клиента. Например, Windows-сервис может вызвать функцию *LogonUser* для запуска нового сеанса входа. Эта функция возвращает маркер доступа, из которого сервис может извлечь SID входа. Потом этот SID сервис может использовать в ACE, разрешающем обращение к интерактивным объектам WindowStation и Desktop из сеанса входа клиента. SID для сеанса входа выглядит как S-1-5-5-0, а RID генерируется случайным образом.

Таблица 8-2. Некоторые общеизвестные SID

SID	Группа	Описание
S-1-1-0	Everyone	Группа, включающая всех пользователей
S-1-2-0	Local	Пользователи, которые регистрируются на терминалах, локально (физически) подключенных к системе
S-1-3-0	Creator Owner ID	Идентификатор защиты, подлежащий замене идентификатором защиты пользователя, создавшего новый объект; этот SID применяется в наследуемых ACE
S-1-3-1	Creator Group ID	Идентификатор защиты, подлежащий замене SID основной группы, в которую входит пользователь, создавший новый объект; этот SID применяется в наследуемых ACE

ЭКСПЕРИМЕНТ: просмотр SID учетных записей с помощью PsGetSid

Представление SID для любой учетной записи легко увидеть, запустив утилиту PsGetSid (*www.sysinternals.com*). У нее следующий интерфейс:

```
C:\>psgetsid -?

PsGetSid displays the machine SID for the local
or remote Windows system.

Usage: psgetsid [\\RemoteComputer [-u Username [-p Password]]]
[account | SID]
     -u       Specifies optional user name for login to
              remote computer.
```

```
-p       Specifies optional password for user name.
         If you omit this you will be prompted to enter
         a hidden password.
account  PsGetSid will report the SID for the specified
         user account rather than the computer.
SID      PsGetSid will report the account for the
         specified SID.
```

Параметры PsGetSid позволяют транслировать имена учетных записей пользователей и компьютеров в соответствующие SID и наоборот.

Если PsGetSid запускается без параметров, она выводит SID, назначенный локальному компьютеру. Используя тот факт, что учетной записи Administrator всегда присваивается RID, равный 500, вы можете определить имя этой учетной записи (в тех случаях, когда системный администратор переименовал ее по соображениям безопасности), просто передав SID компьютера, дополненный «–500», как аргумент командной строки PsGetSid.

Чтобы получить SID доменной учетной записи, введите имя пользователя с указанием домена:

```
c:\>psgetsid redmond\daryl
```

Вы можете выяснить SID домена, передав в PsGetSid его имя как аргумент:

```
c:\>psgetsid Redmond
```

Наконец, изучив RID своей учетной записи, вы по крайней мере узнаете, сколько учетных записей защиты (security accounts), равное значению, полученному вычитанием 999 из вашего RID, было создано в вашем домене или на вашем локальном компьютере (в зависимости от используемой вами учетной записи — доменной или локальной). Чтобы определить, каким учетным записям были присвоены RID, передайте SID с интересующим вас RID. Если PsGetSid сообщит, что сопоставить SID с каким-либо именем учетной записи не удалось, и значение RID окажется меньше, чем у вашей учетной записи, значит, учетная запись, по которой был присвоен RID, уже удалена.

Например, чтобы выяснить имя учетной записи, по которой был назначен двадцать восьмой RID, передайте SID домена с добавлением «–1027»:

```
c:\>psgetsid S-1-5-21-1787744166-3910675280-2727264193-1027
Account for S-1-5-21-1787744166-3910675280-2727264193-1027:
User: redmond\daryl
```

Маркеры

Для идентификации контекста защиты процесса или потока SRM использует объект, называемый *маркером* (token), или *маркером доступа* (access token). В контекст защиты входит информация, описывающая привилегии, учетные записи и группы, сопоставленные с процессом или потоком. В процессе входа в систему (этот процесс рассматривается в конце главы) Winlogon создает начальный маркер, представляющий пользователя, который входит в систему, и сопоставляет его с начальным процессом (или процессами) — по умолчанию запускается Userinit.exe. Так как дочерние процессы по умолчанию наследуют копию маркера своего создателя, все процессы в сеансе данного пользователя выполняются с одним и тем же маркером. Вы можете сгенерировать маркер вызовом Windows-функции *LogonUser* и применить его для создания процесса, который будет выполняться в контексте защиты пользователя, зарегистрированного с помощью функции *LogonUser*; с этой целью вы должны передать полученный маркер Windows-функции *CreateProcessAsUser*. Функция *CreateProcessWithLogon* тоже создает маркер, создавая новый сеанс входа с начальным процессом. Именно так команда *runas* запускает процессы с альтернативными маркерами.

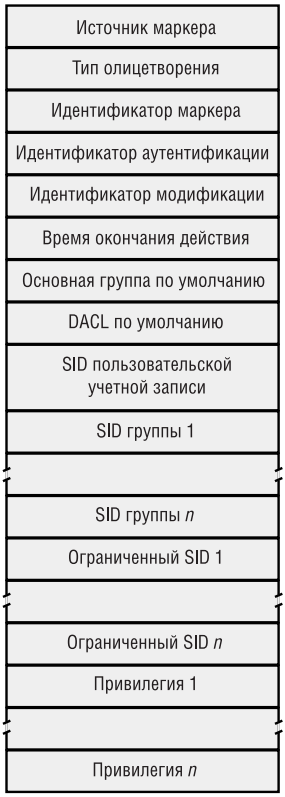


Рис. 8-3. *Маркеры доступа*

Длина маркеров варьируется из-за того, что учетные записи разных пользователей имеют неодинаковые наборы привилегий и сопоставлены с разными учетными записями групп. Но все маркеры содержат одну и ту же информацию, показанную на рис. 8-3.

Механизмы защиты в Windows используют два элемента маркера, определяя, какие объекты доступны и какие операции можно выполнять. Первый элемент состоит из SID учетной записи пользователя и полей SID групп. Используя SID-идентификаторы, SRM определяет, можно ли предоставить запрошенный тип доступа к защищаемому объекту, например к файлу в NTFS.

SID групп в маркере указывают, в какие группы входит учетная запись пользователя. При обработке клиентских запросов серверные приложения могут блокировать определенные группы для ограничения удостоверений защиты, сопоставленных с маркером. Блокирование группы дает почти тот же эффект, что и ее исключение из маркера. (Блокированные SID все же используются при проверке прав доступа, но об этом мы расскажем потом.)

Вторым элементом маркера, определяющим, что может делать поток или процесс, которому назначен данный маркер, является список привилегий — прав, сопоставленных с маркером. Примером привилегии может служить право процесса или потока, сопоставленного с маркером, на выключение компьютера. (Подробнее привилегии будут рассмотрены позже.) Поля основной группы маркера по умолчанию и списка управления избирательным доступом (discretionary access-control list, DACL) представляют собой атрибуты защиты, применяемые Windows к объектам, которые создаются процессом или потоком с использованием маркера. Включая в маркеры информацию о защите, Windows упрощает процессам и потокам создание объектов со стандартными атрибутами защиты, так как в этом случае им не требуется запрашивать информацию о защите при создании каждого объекта.

Маркер может быть основным (primary token) (идентифицирует контекст защиты процесса) и олицетворяющим (impersonation token) (применяется для временного заимствования потоком другого контекста защиты — обычно другого пользователя). Маркеры олицетворения сообщают уровень олицетворения, определяющий, какой тип олицетворения активен в маркере.

Остальные поля маркера служат для информационных нужд. Поле источника маркера содержит сведения (в текстовой форме) о создателе маркера. Оно позволяет различать такие источники, как диспетчер сеансов Windows, сетевой файл-сервер или RPC-сервер. Идентификатор маркера представляет собой локально уникальный идентификатор (locally unique identifier, LUID), который SRM присваивает маркеру при его создании. Исполнительная система поддерживает свой LUID — счетчик, с помощью которого она назначает каждому маркеру уникальный числовой идентификатор.

Еще одна разновидность LUID — идентификатор аутентификации (authentication ID). Он назначается маркеру создателем при вызове функции *LsaLogonUser*. Если создатель не указывает LUID, то LSASS формирует LUID из LUID исполнительной системы. LSASS копирует идентификатор аутентификации для всех маркеров — потомков начального маркера. Используя этот

идентификатор, программа может определить, принадлежит ли какой-то маркер тому же сеансу, что и остальные маркеры, анализируемые данной программой.

LUID исполнительной системы обновляет идентификатор модификации при каждом изменении характеристик маркера. Проверяя этот идентификатор, программа может обнаруживать изменения в контексте защиты с момента его последнего использования.

Маркеры содержат поле времени окончания действия, которое присутствует в них, начиная с Windows NT 3.1, но до сих пор не используется. Будущая версия Windows, возможно, будет поддерживать маркеры, действительные только в течение определенного срока. Представьте, что администратор выдал пользователю учетную запись, срок действия которой ограничен. Сейчас, если срок действия учетной записи истекает в тот момент, когда пользователь все еще находится в системе, он может по-прежнему обращаться к системным ресурсам, доступ к которым был разрешен по просроченной учетной записи. Единственное, что можно сделать в такой ситуации, — принудительно завершить сеанс работы этого пользователя. Если бы Windows поддерживала маркеры с ограниченным сроком действия, система могла бы запретить пользователю доступ к ресурсу сразу по окончании срока действия маркера.

ЭКСПЕРИМЕНТ: просмотр маркеров доступа

Команда *dt _TOKEN* отладчика ядра показывает формат внутреннего объекта «маркер». Хотя его структура отличается от структуры маркера пользовательского режима, возвращаемой Windows-функциями защиты, их поля аналогичны. Детальное описание маркеров см. в документации Platform SDK.

Ниже приведен пример вывода команды *dt _TOKEN* отладчика ядра.

```
kd> dt _TOKEN
   +0x000 TokenSource      : _TOKEN_SOURCE
   +0x010 TokenId          : _LUID
   +0x018 AuthenticationId : _LUID
   +0x020 ParentTokenId    : _LUID
   +0x028 ExpirationTime   : _LARGE_INTEGER
   +0x030 TokenLock        : Ptr32 _ERESOURCE
   +0x034 ModifiedId       : _LUID
   +0x03c SessionId        : Uint4B
   +0x040 UserAndGroupCount : Uint4B
   +0x044 RestrictedSidCount : Uint4B
   +0x048 PrivilegeCount   : Uint4B
   +0x04c VariableLength   : Uint4B
   +0x050 DynamicCharged   : Uint4B
   +0x054 DynamicAvailable : Uint4B
   +0x058 DefaultOwnerIndex : Uint4B
   +0x05c UserAndGroups    : Ptr32 _SID_AND_ATTRIBUTES
   +0x060 RestrictedSids   : Ptr32 _SID_AND_ATTRIBUTES
```

```
   +0x064 PrimaryGroup     : Ptr32 Void
   +0x068 Privileges       : Ptr32 _LUID_AND_ATTRIBUTES
   +0x06c DynamicPart      : Ptr32 Uint4B
   +0x070 DefaultDacl      : Ptr32 _ACL
   +0x074 TokenType        : _TOKEN_TYPE
   +0x078 ImpersonationLevel : _SECURITY_IMPERSONATION_LEVEL
   +0x07c TokenFlags       : UChar
   +0x07d TokenInUse       : UChar
   +0x080 ProxyData        : Ptr32 _SECURITY_TOKEN_PROXY_DATA
   +0x084 AuditData        : Ptr32 _SECURITY_TOKEN_AUDIT_DATA
   +0x088 VariablePart     : Uint4B
```

Маркер для процесса можно увидеть с помощью команды *!token*. Адрес маркера вы найдете в информации, сообщаемой командой *!process*, как показано в следующем примере.

```
kd> !process 380 1
!process 380 1
Searching for Process with Cid == 380
PROCESS ff8027a0 SessionId: 0  Cid: 0380  Peb: 7ffdf000
  ParentCid: 0124  DirBase: 06433000  ObjectTable: ff7e0b68
  TableSize:  23.
  Image: cmd.exe
  VadRoot 84c30568 Clone 0 Private 77. Modified 0. Locked 0.
  DeviceMap 818a3368
  Token                             e22bc730
  ElapsedTime                       14:22:56.0536
  UserTime                          0:00:00.0040
  KernelTime                        0:00:00.0100
  QuotaPoolUsage[PagedPool]         13628
  QuotaPoolUsage[NonPagedPool]      1616
  Working Set Sizes (now,min,max)  (261, 50, 345)(1044KB, 200KB, 1380KB)
  PeakWorkingSetSize                262
  VirtualSize                       11 Mb
  PeakVirtualSize                   11 Mb
  PageFaultCount                    313
  MemoryPriority                    FOREGROUND
  BasePriority                      8
  CommitCharge                      86

kd> !token e22bc730
_TOKEN e22bc730
TS Session ID: 0
User: S-1-5-21-1787744166-3910675280-2727264193-500
Groups:
 00 S-1-5-21-1787744166-3910675280-2727264193-513
    Attributes - Mandatory Default Enabled
 01 S-1-1-0
    Attributes - Mandatory Default Enabled
```

см. след. стр.

```
 02 S-1-5-32-544
    Attributes - Mandatory Default Enabled Owner
 03 S-1-5-32-545
    Attributes - Mandatory Default Enabled
 04 S-1-5-5-0-92587
    Attributes - Mandatory Default Enabled LogonId
 05 S-1-2-0
    Attributes - Mandatory Default Enabled
 06 S-1-5-14
    Attributes - Mandatory Default Enabled
 07 S-1-5-4
    Attributes - Mandatory Default Enabled
 08 S-1-5-11
    Attributes - Mandatory Default Enabled
Primary Group: S-1-5-21-1787744166-3910675280-2727264193-513
Privs:
 00 0x000000017 SeChangeNotifyPrivilege           Attributes -
 Enabled Default
 01 0x000000008 SeSecurityPrivilege               Attributes -
 02 0x000000011 SeBackupPrivilege                 Attributes -
 03 0x000000012 SeRestorePrivilege                Attributes -
 04 0x00000000c SeSystemtimePrivilege             Attributes -
 05 0x000000013 SeShutdownPrivilege               Attributes -
 06 0x000000018 SeRemoteShutdownPrivilege         Attributes -
 07 0x000000009 SeTakeOwnershipPrivilege          Attributes -
 08 0x000000014 SeDebugPrivilege                  Attributes -
 09 0x000000016 SeSystemEnvironmentPrivilege      Attributes -
 10 0x00000000b SeSystemProfilePrivilege          Attributes -
 11 0x00000000d SeProfileSingleProcessPrivilege   Attributes -
 12 0x00000000e SeIncreaseBasePriorityPrivilege   Attributes -
 13 0x00000000a SeLoadDriverPrivilege             Attributes -
 Enabled
 14 0x00000000f SeCreatePagefilePrivilege         Attributes -
 15 0x000000005 SeIncreaseQuotaPrivilege          Attributes -
 16 0x000000019 SeUndockPrivilege                 Attributes -
 Enabled
 17 0x00000001c SeManageVolumePrivilege           Attributes -
Authentication ID:         (0,169e4)
Impersonation Level:       Anonymous
TokenType:                 Primary
Source: User32             TokenFlags: 0x9 ( Token in use )
Token ID: 3a86d0d          ParentToken ID: 0
Modified ID:               (0, 3a86d0f)
RestrictedSidCount: 0      RestrictedSids: 00000000
```

Содержимое маркера можно косвенно увидеть с помощью Process Explorer (*www.sysinternals.com*) на вкладке Security в диалоговом окне свойств процесса. В этом окне показываются группы и привилегии, включенные в маркер исследуемого вами процесса.

Олицетворение

Олицетворение (impersonation) — мощное средство, часто используемое в модели защиты Windows. Олицетворение также применяется в модели программирования «клиент-сервер». Например, серверное приложение может экспортировать ресурсы (файлы, принтеры или базы данных). Клиенты, которые хотят обратиться к этим ресурсам, посылают серверу запрос. Получив запрос, сервер должен убедиться, что у клиента есть разрешение на выполнение над ресурсом запрошенных операций. Так, если пользователь на удаленной машине пытается удалить файл с сетевого диска NTFS, сервер, экспортирующий этот сетевой ресурс, должен проверить, имеет ли пользователь право удалить данный файл. Казалось бы, в таком случае сервер должен запросить учетную запись пользователя и SID-идентификаторы группы, а также просканировать атрибуты защиты файла. Но этот процесс труден для программирования, подвержен ошибкам и не позволяет обеспечить поддержку новых функций защиты. Поэтому Windows в таких ситуациях предоставляет серверу сервисы олицетворения.

Олицетворение позволяет серверу уведомить SRM о временном заимствовании профиля защиты клиента, запрашивающего ресурс. После этого сервер может обращаться к ресурсам от имени клиента, а SRM — проводить проверку его прав доступа. Обычно серверу доступен более широкий круг ресурсов, чем клиенту, и при олицетворении сервер может терять часть исходных прав доступа. Также вероятно и обратное: при олицетворении сервер может получить дополнительные права.

Сервер олицетворяет клиент лишь в пределах потока, выдавшего запрос на олицетворение. Управляющие структуры данных потока содержат необязательный элемент для маркера доступа. Однако основной маркер потока, отражающий его реальные права, всегда доступен через управляющие структуры процесса.

За поддержку олицетворения в Windows отвечает несколько механизмов. Если сервер взаимодействует с клиентом через именованный канал, он может вызвать Windows-функцию *ImpersonateNamedPipeClient* и тем самым сообщить SRM о том, что ему нужно подменить собой пользователя на другом конце канала. Если сервер взаимодействует с клиентом через DDE (Dynamic Data Exchange) или RPC, то выдает аналогичный запрос на олицетворение через *DdeImpersonateClient* или *RpcImpersonateClient*. Поток может создать маркер олицетворения просто как копию маркера своего процесса, вызвав функцию *ImpersonateSelf*. Для блокировки каких-то SID или привилегий поток может потом изменить полученный маркер олицетворения. Наконец, пакет SSPI (Security Support Provider Interface) может олицетворять своих клиентов через *ImpersonateSecurityContext*. SSPI реализует модель сетевой защиты вроде LAN Manager версии 2 или Kerberos.

После того как серверный поток завершает выполнение своей задачи, он возвращает себе прежний профиль защиты. Эти формы олицетворения удобны для выполнения определенных операций по запросу клиента и для корректного аудита обращений к объектам. (Например, генерируемые данные

аудита сообщают идентификацию подменяемого клиента, а не серверного процесса.) Их недостаток в том, что нельзя выполнять всю программу в контексте клиента. Кроме того, маркер олицетворения не дает доступа к сетевым файлам или принтерам, если только они не поддерживают null-сеансы или не используется олицетворение уровня делегирования (delegation-level impersonation), причем удостоверения защиты достаточны для аутентификации на удаленном компьютере. (Null-сеанс создается при анонимном входе.)

Если все приложение должно выполняться в контексте защиты клиента или получать доступ к сетевым ресурсам, клиент должен быть зарегистрирован в системе. Для этого предназначена Windows-функция *LogonUser*, которая принимает в качестве параметров имя учетной записи, пароль, имя домена или компьютера, тип входа (интерактивный, пакетный или сервисный) и провайдер входа (logon provider), а возвращает основной маркер. Серверный поток принимает маркер в виде маркера олицетворения, либо сервер запускает программу, основной маркер которой включает удостоверения клиента. С точки зрения защиты, процесс, создаваемый с применением маркера, который возвращается при интерактивном входе через *LogonUser*, например API-функцией *CreateProcessAsUser*, выглядит как программа, запущенная пользователем при интерактивном входе в систему. Недостаток этого подхода в том, что серверу приходится получать имя и пароль по учетной записи пользователя. Если сервер передает эту информацию по сети, он должен надежно шифровать ее, чтобы избежать получения имени и пароля злоумышленником, перехватывающим сетевой трафик.

Windows не позволяет серверам подменять клиенты без их ведома. Клиентский процесс может ограничить уровень олицетворения серверным процессом, сообщив при соединении с ним требуемый SQOS (Security Quality of Service). Процесс может указывать флаги SECURITY_ANONYMOUS, SECURITY_IDENTIFICATION, SECURITY_IMPERSONATION и SECURITY_DELEGATION при вызове Windows-функции *CreateFile*. Каждый уровень позволяет серверу выполнять различный набор операций относительно контекста защиты клиента:

- SecurityAnonymous — самый ограниченный уровень; сервер не может олицетворять или идентифицировать клиент;
- SecurityIdentification — сервер может получать SID и привилегии клиента, но не получает право на олицетворение клиента;
- SecurityImpersonation — сервер может идентифицировать и олицетворять клиент в локальной системе;
- SecurityDelegation — наименее ограниченный уровень. Позволяет серверу олицетворять клиент в локальных и удаленных системах. Windows NT 4 и более ранние версии лишь частично поддерживают этот уровень олицетворения.

Если клиент не устанавливает уровень олицетворения, Windows по умолчанию выбирает SecurityImpersonation. Функция *CreateFile* также принимает модификаторы SECURITY_EFFECTIVE_ONLY и SECURITY_CONTEXT_TRACKING.

Первый из них не дает серверу включать/выключать какие-то привилегии или группы клиента на время олицетворения. А второй указывает, что все изменения, вносимые клиентом в свой контекст защиты, отражаются и на сервере, который олицетворяет этот клиент. Данный модификатор действует, только если клиентский и серверный процессы находятся в одной системе.

Ограниченные маркеры

Ограниченный маркер (restricted token) создается на базе основного или олицетворяющего с помощью функции *CreateRestrictedToken* и является его копией, в которую можно внести следующие изменения:

- удалить некоторые элементы из таблицы привилегий маркера;
- пометить SID-идентификаторы маркера атрибутом проверки только на запрет (deny-only);
- пометить SID-идентификаторы маркера как ограниченные.

Поведение SID с атрибутом проверки только на запрет (deny-only SID) и ограниченных SID (restricted SID) кратко поясняется в следующих разделах. Ограниченные маркеры удобны, когда приложение подменяет клиент при выполнении небезопасного кода. В ограниченном маркере может, например, отсутствовать привилегия на перезагрузку системы, что не позволит коду, выполняемому в контексте защиты ограниченного маркера, перезагрузить систему.

ЭКСПЕРИМЕНТ: просмотр ограниченных маркеров

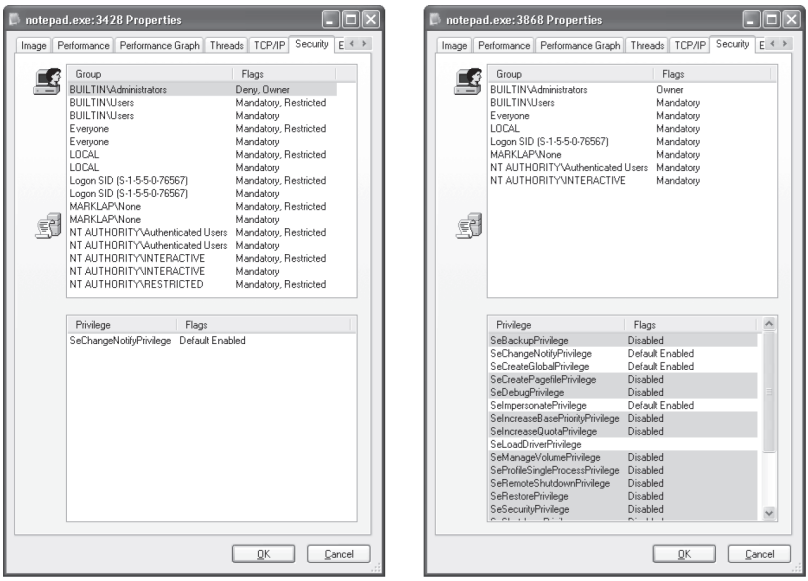
В Windows XP или Windows Server 2003 можно заставить Explorer создать процесс с ограниченным маркером по следующей процедуре.

1. Создайте на рабочем столе ярлык для \Windows\Notepad.exe.
2. Отредактируйте свойства ярлыка и установите флажок Run With Different Credentials (Запускать с другими учетными данными). Заметьте: в описании под этим флажком говорится о том, что вы можете запускать программу от своего имени, в то же время защищая компьютер от несанкционированных действий данной программы*.
3. Закройте окно свойств и запустите программу двойным щелчком ее ярлыка.
4. Согласитесь с параметрами по умолчанию для выполнения под текущей учетной записью и защиты компьютера от несанкционированных действий этой программы.
5. Запустите Process Explorer и просмотрите содержимое вкладки Security для свойств запущенного вами процесса Notepad. Заметьте, что маркер содержит ограниченные SID и SID с атрибутом проверки только на запрет, а также что у него лишь одна привилегия. Свойства в левой части окна, показанного на следующей иллюст-

см. след. стр.

* В русской версии Windows XP ошибочно говорится о защите от несанкционированных действий других программ. — *Прим. перев.*

рации, относятся к Notepad, выполняемому с неограниченным маркером, а свойства в правой части окна — к его экземпляру, запущенному по описанной процедуре.



Ограниченный маркер дает несколько побочных эффектов.

- Удаляются все привилегии, кроме SeChangeNotifyPrivilege.
- Любые SID администраторов или пользователей с правами администраторов помечаются как Deny-Only (проверка только на запрет). Такой SID удаляет права доступа к любым ресурсам, доступ к которым для администраторов запрещен соответствующим ACE, но в ином случае был бы замещен ACE, ранее выданным группе администраторов через дескриптор защиты.
- RESTRICTED SID добавляется в список ограниченных SID, как и все остальные SID маркера, кроме SID пользователя и любых SID администраторов или пользователей с правами администраторов.
- SID учетной записи, под которой вы запустили процесс, не включается в список как ограниченный. То есть процесс не сможет обращаться к объектам, доступ к которым разрешен по вашей учетной записи, но не по учетным записям любых групп, в которые вы входите. Например, у каталога вашего профиля в \Documents and Settings имеется дескриптор защиты по умолчанию, разрешающий доступ по вашей учетной записи, по учетной записи группы администраторов и по учетной записи System. Попытавшись открыть этот каталог из Notepad, запущенного так, как было показано ранее, вы не получите к нему доступа, потому что вторая, внутренняя проверка прав доступа, выполняемая с применением ограниченных SID, закончится неудачей — SID пользователя нет в списке ограниченных SID.

Дескрипторы защиты и управление доступом

Маркеры, которые идентифицируют удостоверения пользователя, являются лишь частью выражения, описывающего защиту объектов. Другая его часть — информация о защите, сопоставленная с объектом и указывающая, кому и какие действия разрешено выполнять над объектом. Структура данных, хранящая эту информацию, называется *дескриптором защиты* (security descriptor). Дескриптор защиты включает следующие атрибуты.

- **Номер версии** Версия модели защиты SRM, использованной для создания дескриптора.
- **Флаги** Необязательные модификаторы, определяющие поведение или характеристики дескриптора. Пример — флаг SE_DACL_PROTECTED, который запрещает наследование дескриптором параметров защиты от другого объекта.
- **SID владельца** Идентификатор защиты владельца.
- **SID группы** Идентификатор защиты основной группы для данного объекта (используется только POSIX).
- **Список управления избирательным доступом (discretionary access-control list, DACL)** Указывает, кто может получать доступ к объекту и какие виды доступа.
- **Системный список управления доступом (system access-control list, SACL)** Указывает, какие операции и каких пользователей должны регистрироваться в журнале аудита безопасности.

Список управления доступом (access-control list, ACL) состоит из заголовка и может содержать элементы (access-control entries, ACE). Существует два типа ACL: DACL и SACL. В DACL каждый ACE содержит SID и маску доступа (а также набор флагов), причем ACE могут быть четырех типов: «доступ разрешен» (access allowed), «доступ отклонен» (access denied), «разрешенный объект» (allowed-object) и «запрещенный объект» (denied-object). Как вы, наверное, и подумали, первый тип ACE разрешает пользователю доступ к объекту, а второй — отказывает в предоставлении прав, указанных в маске доступа.

Разница между ACE типа «разрешенный объект» и «доступ разрешен», а также между ACE типа «запрещенный объект» и «доступ отклонен» заключается в том, что эти типы используются только в Active Directory. ACE этих типов имеют поле глобально уникального идентификатора (globally unique identifier, GUID), которое сообщает, что данный ACE применим только к определенным объектам или подобъектам (с GUID-идентификаторами). Кроме того, необязательный GUID указывает, что тип дочернего объекта наследует ACE при его (объекта) создании в контейнере Active Directory, к которому применен ACE. (GUID — это гарантированно уникальный 128-битный идентификатор.)

За счет аккумуляции прав доступа, сопоставленных с индивидуальными ACE, формируется набор прав, предоставляемых ACL-списком. Если в дескрипторе защиты нет DACL (DACL = null), любой пользователь получает пол-

ный доступ к объекту. Если DACL пуст (т. е. в нем нет ACE), доступа к объекту не получает никто.

ACE, используемые в DACL, также имеют набор флагов, контролирующих и определяющих характеристики ACE, связанные с наследованием. Некоторые пространства имен объектов содержат объекты-контейнеры и объекты-листы (leaf objects). Контейнер может включать другие контейнеры и листы, которые являются его дочерними объектами. Примеры контейнеров — каталоги в пространстве имен файловой системы и разделы в пространстве имен реестра. Отдельные флаги контролируют, как ACE применяется к дочерним объектам контейнера, сопоставленного с этим ACE. Часть правил наследования ACE представлена в таблице 8-3 (полный список см. в Platform SDK).

Таблица 8-3. *Битовые флаги, определяющие правила наследования ACE*

Флаг	Правило наследования
CONTAINER_INHERIT_ACE	Дочерние объекты-контейнеры, например каталоги, наследуют ACE как эффективный (effective ACE). Унаследованный ACE является наследуемым, если не установлен и флаг NO_PROPAGATE_INHERIT_ACE
INHERIT_ONLY_ACE	Указывает, что это только наследуемый ACE (inherit-only ACE); он не контролирует доступ к объекту, сопоставленному с ним
INHERITED_ACE	Указывает, что данный ACE унаследован. Система устанавливает этот флаг при передаче дочернему объекту наследуемого ACE
NO_PROPAGATE_ INHERIT_ACE	Если ACE наследуется дочерним объектом, система сбрасывает флаги OBJECT_INHERIT_ACE и CONTAINER_INHERIT_ACE в унаследованном ACE. Это предотвращает наследование данного ACE последующими поколениями объектов
OBJECT_INHERIT_ACE	Дочерние объекты, не являющиеся контейнерами, наследуют ACE как эффективный. Но дочерние объекты-контейнеры наследуют ACE как «только наследуемый», если не установлен и флаг NO_PROPAGATE_INHERIT_ACE

SACL состоит из ACE двух типов: системного аудита (system audit ACE) и объекта системного аудита (system audit-object ACE). Эти ACE определяют, какие операции, выполняемые над объектами конкретными пользователями или группами, подлежат аудиту. Информация аудита хранится в системном журнале аудита. Аудиту могут подлежать как успешные, так и неудачные операции. Как и специфические для объектов ACE из DACL, ACE объектов системного аудита содержат GUID, указывающий типы объектов или подобъектов, к которым применим данный ACE, и необязательный GUID, контролирующий передачу ACE дочерним объектам конкретных типов. При SACL, равном null, аудит объекта не ведется. (Об аудите безопасности мы расскажем позже.) Флаги наследования, применимые к DACL ACE, применимы к ACE системного аудита и объектов системного аудита.

Упрощенная схема объекта «файл» и его DACL представлена на рис. 8-4.

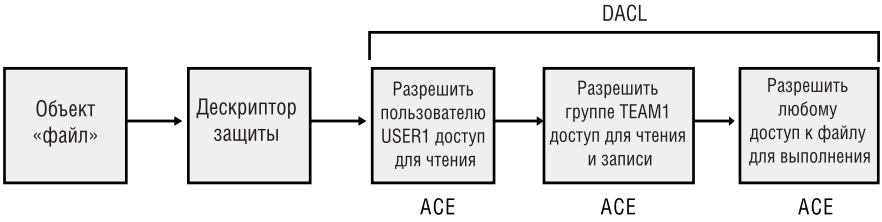


Рис. 8-4. *Список управления избирательным доступом*

Как показано на рис. 8-4, первый ACE позволяет USER1 читать файл. Второй ACE разрешает членам группы TEAM1 читать и записывать файл. Третий ACE предоставляет доступ к файлу для выполнения всем пользователям.

ЭКСПЕРИМЕНТ: просмотр дескриптора защиты

Управляя дескрипторами защиты своих объектов, большинство подсистем исполнительной системы полагаются на функции защиты по умолчанию, предоставляемые диспетчером объектов. Эти функции сохраняют дескрипторы защиты для таких объектов, используя указатель дескриптора защиты (security descriptor pointer). Например, защитой по умолчанию пользуется диспетчер процессов, поэтому диспетчер объектов хранит дескрипторы защиты процессов и потоков в заголовках объектов «процесс» и «поток» соответственно. Указатель дескриптора защиты также применяется для хранения дескрипторов защиты событий, мьютексов и семафоров. Для просмотра дескрипторов защиты этих объектов можно использовать отладчик ядра, но сначала вы должны найти заголовок нужного объекта. Вся эта процедура поясняется ниже.

1. Запустите отладчик ядра.
2. Введите **!process 0 0**, чтобы увидеть адрес Winlogon. (Если в системе активно более одного сеанса Terminal Server, выполняется несколько экземпляров Winlogon.) Затем вновь введите **!process**, но укажите адрес одного из процессов Winlogon:

```
kd> !process 88d0c020
PROCESS 88d0c020  SessionId: 0  Cid: 0bdc    Peb: 7ffdf000
    ParentCid: 00a8   DirBase: 59a58000  ObjectTable: e21df920
    HandleCount:  62.
    Image: winlogon.exe
```

3. Введите **!object** и адрес, следующий за словом PROCESS в выводе предыдущей команды. Это позволит увидеть структуру данных объекта:

```
kd> !object 88d0c020
Object: 88d0c020  Type: (8a0ed388) Process
    ObjectHeader: 88d0c008
    HandleCount: 1  PointerCount: 32
```

см. след. стр.

4. Введите **dt _OBJECT_HEADER** и адрес поля заголовка объекта из вывода предыдущей команды для просмотра структуры данных заголовка объекта, включая значение указателя дескриптора защиты:

```
lkd> dt _OBJECT_HEADER 88d0c008
   +0x000 PointerCount     : 36
   +0x004 HandleCount      : 1
   +0x004 NextToFree       : 0x00000001
   +0x008 Type             : 0x8a0ed388
   +0x00c NameInfoOffset   : 0 ''
   +0x00d HandleInfoOffset : 0 ''
   +0x00e QuotaInfoOffset  : 0 ''
   +0x00f Flags            : 0x20 ''
   +0x010 ObjectCreateInfo : 0x89e596a0
   +0x010 QuotaBlockCharged : 0x89e596a0
   +0x014 SecurityDescriptor : 0xe242b864
   +0x018 Body             : _QUAD
```

5. Указатели дескрипторов защиты в заголовке объекта используют младшие три бита как флаги, поэтому следующая команда позволяет создать дамп дескриптора защиты. Вы указываете адрес, полученный из структуры заголовка объекта, но удаляете его младшие три бита:

```
lkd> !sd 0xe242b864 & -8
->Revision: 0x1
->Sbz1    : 0x0
->Control : 0x8004
            SE_DACL_PRESENT
            SE_SELF_RELATIVE
->Owner   : S-1-5-21-1787744166-3910675280-2727264193-500
->Group   : S-1-5-21-1787744166-3910675280-2727264193-513
->Dacl    :
->Dacl    : ->AclRevision: 0x2
->Dacl    : ->Sbz1       : 0x0
->Dacl    : ->AclSize    : 0x40
->Dacl    : ->AceCount   : 0x2
->Dacl    : ->Sbz2       : 0x0
->Dacl    : ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[0]: ->AceFlags: 0x0
->Dacl    : ->Ace[0]: ->AceSize: 0x24
->Dacl    : ->Ace[0]: ->Mask : 0x001f0fff
->Dacl    : ->Ace[0]: ->SID: S-1-5-21-1787744166-
3910675280-2727264193-500

->Dacl    : ->Ace[1]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[1]: ->AceFlags: 0x0
->Dacl    : ->Ace[1]: ->AceSize: 0x14
->Dacl    : ->Ace[1]: ->Mask : 0x001f0fff
->Dacl    : ->Ace[1]: ->SID: S-1-5-18

->Sacl    :  is NULL
```

Дескриптор защиты содержит два ACE типа «доступ разрешен», причем один из них указывает учетную запись администратора (ее можно распознать по RID, равному 500), а другой — учетную запись System (которая всегда выглядит как S-1-5-18). Без декодирования битов, установленных в масках доступа в ACE и определения того, каким типам доступа к процессам они соответствуют, очень трудно сказать, какими правами доступа к объекту «процесс» для Winlogon обладает каждая из этих учетных записей. Однако, если вы сделаете это, используя заголовочные файлы из SDK, то обнаружите, что обе учетные записи имеют полные права доступа.

Присвоение ACL

Чтобы определить, какой DACL следует назначить новому объекту, система защиты использует первое применимое правило из следующего списка.

1. Если вызывающий поток явно предоставляет дескриптор защиты при создании объекта, то система защиты применяет его к объекту. Если у объекта есть имя и он находится в объекте-контейнере (например, именованное событие в каталоге \BaseNamedObjects пространства имен диспетчера объектов), система объединяет в DACL все наследуемые ACE (ACE, которые могут быть переданы от контейнера объекта), но только в том случае, если в дескрипторе защиты не установлен флаг SE_DACL_PROTECTED, запрещающий наследование.
2. Если вызывающий поток не предоставляет дескриптор защиты и объекту присваивается имя, система защиты ищет этот дескриптор в контейнере, в котором хранится имя нового объекта. Некоторые ACE каталога объектов могут быть помечены как наследуемые. Это означает, что они должны применяться к новым объектам, создаваемым в данном каталоге. При наличии наследуемых ACE система защиты формирует из них ACL, назначаемый новому объекту. (В ACE, наследуемых только объектами-контейнерами, устанавливаются отдельные флаги.)
3. Если дескриптор защиты не определен и объект не наследует какие-либо ACE, система защиты извлекает DACL по умолчанию из маркера доступа вызывающего потока и применяет его к новому объекту. В некоторые подсистемы Windows (например, службы, LSA и SAM-объекты) «зашиты» свои DACL, назначаемые ими объектам при создании.
4. Если дескриптор защиты не определен и нет ни наследуемых ACE, ни DACL по умолчанию, система создает объект без DACL, что открывает полный доступ к нему любым пользователям и группам. Это правило идентично третьему, если маркер содержит нулевой DACL по умолчанию.

Правила, используемые системой при назначении SACL новому объекту, аналогичны правилам присвоения DACL за двумя исключениями. Первое заключается в том, что наследуемые ACE системного аудита не передаются

объектам с дескрипторами защиты, помеченными флагом SE_SACL_PROTECTED (DACL точно так же защищается флагом SE_DACL_PROTECTED). Второе исключение: если ACE системного аудита не определены и наследуемого SACL нет, то SACL вообще не присваивается объекту (в маркерах нет SACL по умолчанию).

Когда к контейнеру применяется новый дескриптор защиты, содержащий наследуемые ACE, система автоматически передает их в дескрипторы защиты дочерних объектов. (Заметьте, что DACL дескриптора защиты не принимает наследуемые DACL ACE, если установлен флаг SE_DACL_PROTECTED, а его SACL не наследует SACL ACE, если установлен флаг SE_SACL_PROTECTED.) В соответствии с порядком слияния наследуемых ACE с дескриптором защиты дочернего объекта любые ACE, явно примененные к ACL, размещаются до ACE, унаследованных объектом. Система использует следующие правила передачи наследуемых ACE.

- Если дочерний объект без DACL наследует ACE, он получает DACL, содержащий лишь унаследованные ACE.
- Если дочерний объект с пустым DACL наследует ACE, он также получает DACL, содержащий лишь унаследованные ACE.
- Только для объектов в Active Directory: если наследуемый ACE удаляется из родительского объекта, все копии этого ACE автоматически удаляются из всех дочерних объектов.
- Только для объектов в Active Directory: если из DACL дочернего объекта автоматически удалены все ACE, у дочернего объекта остается пустой DACL.

Как вы вскоре убедитесь, порядок ACE в ACL является важным аспектом модели защиты Windows.

> **ПРИМЕЧАНИЕ** Как правило, наследование не поддерживается напрямую такими хранилищами объектов, как файловые системы, реестр или Active Directory. Функции Windows API, поддерживающие наследование, в том числе *SetSecurityInfo* и *SetNamedSecurityInfo*, реализуют наследование вызовом соответствующих функций из DLL поддержки наследования атрибутов защиты (\Windows\System32\Ntmarta.Dll), которым известно, как устроены эти хранилища объектов.

Определение прав доступа

Для определения прав доступа к объекту используются два алгоритма:

- сравнивающий запрошенные права с максимально возможными для данного объекта и экспортируемый в пользовательский режим в виде Windows-функции *GetEffectiveRightsFromAcl*;
- проверяющий наличие конкретных прав доступа и активизируемый через Windows-функцию *AccessCheck* или *AccessCheckByType*.

Первый алгоритм проверяет элементы DACL следующим образом.

1. В отсутствие DACL (DACL = null) объект является незащищенным, и система защиты предоставляет к нему полный доступ.
2. Если у вызывающего потока имеется привилегия на захват объекта во владение (take-ownership privilege), система защиты предоставляет владельцу право на доступ для записи (write-owner access) до анализа DACL (что такое привилегия захвата объекта во владение и право владельца на доступ для записи, мы поясним чуть позже).
3. Если вызывающий поток является владельцем объекта, ему предоставляются права управления чтением (read-control access) и доступа к DACL для записи (write-DACL access).
4. Из маски предоставленных прав доступа удаляется маска доступа каждого ACE типа «доступ отклонен», SID которого совпадает с SID маркера доступа вызывающего потока.
5. К маске предоставленных прав доступа добавляется маска доступа каждого ACE типа «доступ разрешен», SID которого совпадает с SID маркера доступа вызывающего потока (исключение составляют права доступа, в предоставлении которых уже отказано).

После анализа всех элементов DACL рассчитанная маска предоставленных прав доступа возвращается вызывающему потоку как максимальные права доступа. Эта маска отражает полный набор типов доступа, которые этот поток сможет успешно запрашивать при открытии данного объекта.

Все сказанное применимо лишь к той разновидности алгоритма, которая работает в режиме ядра. Его Windows-версия, реализованная функцией *GetEffectiveRightsFromAcl*, отличается отсутствием шага 2, а также тем, что вместо маркера доступа она рассматривает SID единственного пользователя или группы.

Второй алгоритм проверяет, можно ли удовлетворить конкретный запрос на доступ, исходя из маркера доступа вызывающего потока. У каждой Windows-функции открытия защищенных объектов есть параметр, указывающий желательную маску доступа — последний элемент выражения, описывающего защиту объектов. Чтобы определить, имеет ли вызывающий поток право на доступ к защищенному объекту, выполняются следующие операции.

1. В отсутствие DACL (DACL = null) объект является незащищенным, и система защиты предоставляет к нему запрошенный тип доступа.
2. Если у вызывающего потока имеется привилегия на захват объекта во владение, система защиты предоставляет владельцу право на доступ для записи, а затем анализирует DACL. Однако, если такой поток запросил только доступ владельца для записи, система защиты предоставляет этот тип доступа и не просматривает DACL.
3. Если вызывающий поток является владельцем объекта, ему предоставляются права управления чтением и доступа к DACL для записи. Если вызывающий поток запросил только эти права, система защиты предоставляет их без просмотра DACL.

4. Просматриваются все ACE в DACL — от первого к последнему. Обработка ACE выполняется при одном из следующих условий:
   a. SID в ACE типа «доступ отклонен» совпадает с незаблокированным SID (SID могут быть незаблокированными и заблокированными) или SID с атрибутом проверки только на запрет в маркере доступа вызывающего потока;
   b. SID в ACE типа «доступ разрешен» совпадает с незаблокированным SID в маркере доступа вызывающего потока, и этот SID не имеет атрибута проверки только на запрет;
   c. Идет уже второй проход поиска в дескрипторе ограниченных SID, и SID в ACE совпадает с ограниченным SID в маркере доступа вызывающего потока.
5. В случае ACE типа «доступ разрешен» предоставляются запрошенные права из маски доступа ACE; проверка считается успешной, если предоставляются все запрошенные права. Доступ к объекту не предоставляется в случае ACE типа «доступ отклонен» и отказа в предоставлении какого-либо из запрошенных прав.
6. Если достигнут конец DACL и некоторые из запрошенных прав доступа еще не предоставлены, доступ к объекту запрещается.
7. Если все права доступа предоставлены, но в маркере доступа вызывающего потока имеется хотя бы один ограниченный SID, то система повторно сканирует DACL в поисках ACE, маски доступа которых соответствуют набору запрошенных прав доступа. При этом также идет поиск ACE, SID которых совпадает с любым из ограниченных SID вызывающего потока. Поток получает доступ к объекту, если запрошенные права доступа предоставлялись после каждого прохода по DACL.

Поведение обоих алгоритмов проверки прав доступа зависит от относительного расположения разрешающих и запрещающих ACE. Возьмем для примера объект с двумя ACE, первый из которых указывает, что определенному пользователю разрешен полный доступ к объекту, а второй отказывает в доступе. Если разрешающий ACE предшествует запрещающему, пользователь получит полный доступ к объекту. При другом порядке этих ACE пользователь вообще не получит доступа к объекту.

Более старые Windows-функции вроде *AddAccessAllowedAce* добавляли ACE в конец DACL, что нежелательно. Таким образом, до появления Windows 2000 большинство Windows-приложений были вынуждены создавать DACL вручную, помещая запрещающие ACE в начало списка. Несколько функций Windows, например *SetSecurityInfo* и *SetNamedSecurityInfo*, используют предпочтительный порядок ACE: запрещающие ACE предшествуют разрешающим. Заметьте, что эти функции вызываются при редактировании, например, прав доступа к NTFS-файлам и разделам реестра. *SetSecurityInfo* и *SetNamedSecurityInfo* также применяют правила наследования ACE к дескриптору защиты, для операций над которым они вызываются.

На рис. 8-5 показан пример проверки прав доступа, демонстрирующий, насколько важен порядок ACE. В этом примере пользователю отказано в доступе к файлу, хотя ACE в DACL объекта предоставляет такое право (в силу принадлежности пользователя к группе Writers). Это вызвано тем, что запрещающий ACE предшествует разрешающему.

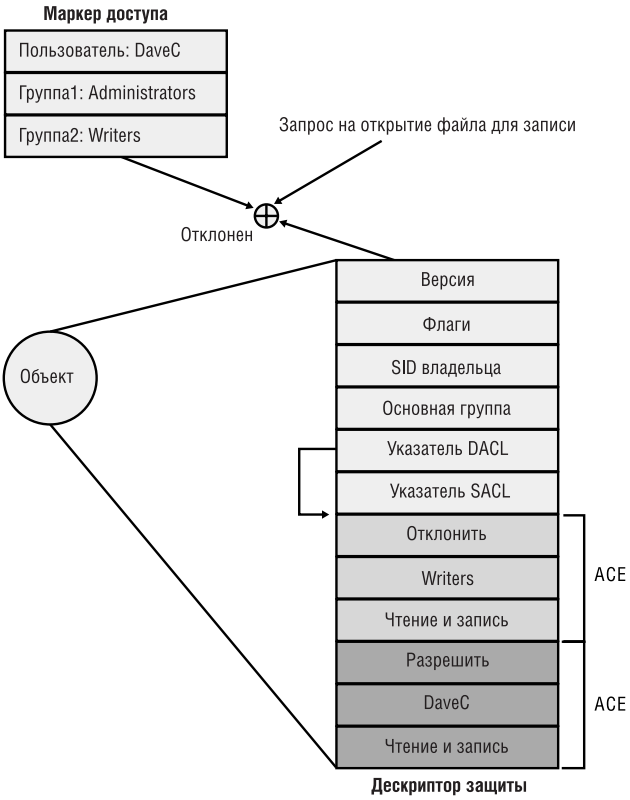


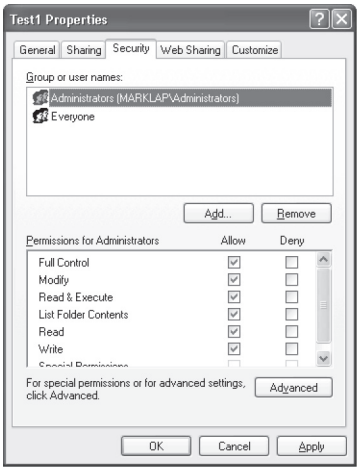
Рис. 8-5. *Пример проверки прав доступа*

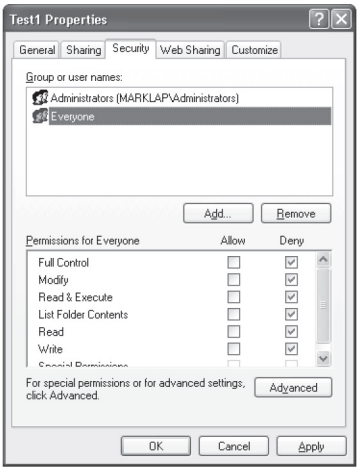
Как уже говорилось, обработка DACL системой защиты при каждом использовании описателя процессом была бы неэффективной, поэтому SRM проверяет права доступа только при открытии описателя, а не при каждом его использовании. Так что, если процесс один раз успешно открыл описатель, система защиты не может аннулировать предоставленные при этом права доступа — даже когда DACL объекта изменяется. Учтите и вот еще что: поскольку код режима ядра обращается к объектам по указателям, а не по описателям, при использовании объектов операционной системой права доступа не проверяются. Иначе говоря, исполнительная система полностью доверяет себе в смысле защиты.

Тот факт, что владелец объекта всегда получает право на запись DACL при доступе к объекту, означает, что пользователям нельзя запретить доступ к принадлежащим им объектам. Если в силу каких-то причин DACL объекта пуст (доступ запрещен), владелец все равно может открыть объект с правом записи DACL и применить новый DACL, определяющий нужные права доступа.

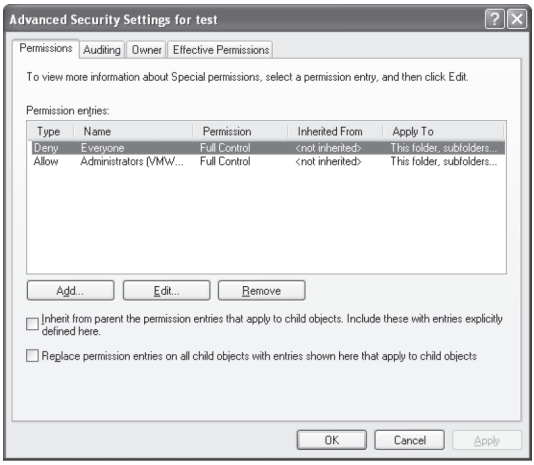
Будьте осторожны при использовании GUI-средств изменения параметров защиты

Модифицируя с помощью GUI-средств параметры защиты объектов «файл», «реестр», Active Directory или других защищаемых объектов, имейте в виду, что основное диалоговое окно безопасности создает потенциально неверное представление о защите, применяемой для объекта. В верхней части этого окна в алфавитном порядке показываются группы и пользователи, чьи ACE имеются в ACL данного объекта. Если вы выдадите Full Control группе Administrators и запретите его группе Everyone, то, судя по алфавитному списку, можете подумать, будто ACE типа «доступ разрешен» для группы Administrators предшествует ACE типа «доступ отклонен» для группы Everyone. Однако, как мы уже говорили, средства редактирования, применяя ACL к объекту, помещают запрещающие ACE перед разрешающими.

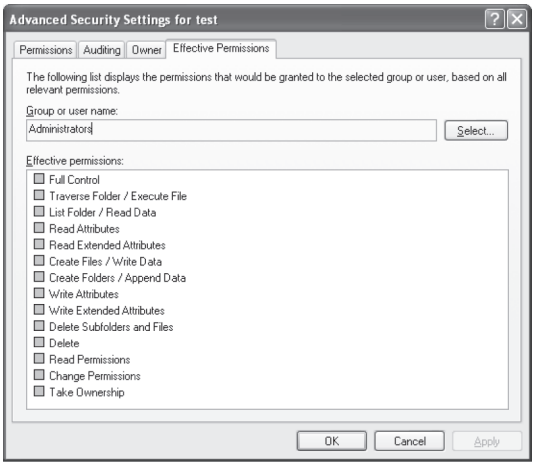




На вкладке Permissions (Разрешения) диалогового окна Advanced Security Settings (Дополнительные параметры безопасности) показывается порядок ACE в DACL. Однако даже это диалоговое окно может ввести в заблуждение, так как в сложном DACL за запрещающими ACE для различных видов доступа могут быть расположены разрешающие ACE для других типов доступа.



Единственный способ точно узнать, какие виды доступа к объекту будут разрешены конкретному пользователю или группе (помимо метода проб и ошибок), — открыть вкладку Effective Permissions. Введите здесь имя пользователя или группы, и диалоговое окно покажет, какие разрешения на доступ к объекту будут действовать на самом деле.



AuthZ API

AuthZ API, впервые введенный в Windows XP, реализует ту же модель защиты, что и Security Reference Monitor (монитор состояния защиты), но исключительно для пользовательского режима; все функции AuthZ API находятся в библиотеке \Windows\System32\Authz.Dll. Это позволяет приложениям, нуждающимся в защите своих закрытых объектов (вроде таблиц базы данных), задействовать Windows-модель защиты без издержек, связанных с переходами из пользовательского режима

см. след. стр.

в режим ядра, которые были бы неизбежны при использовании Security Reference Monitor.

AuthZ API оперирует стандартными структурами дескриптора защиты, SID и привилегиями. Вместо применения маркеров для представления клиентов, AuthZ использует AUTHZ_CLIENT_CONTEXT. AuthZ включает эквиваленты всех функций проверки прав доступа и защиты Windows; например, *AuthzAccessCheck* — это AuthZ-версия Windows-функции *AccessCheck*, которая вызывает функцию *SeAccessCheck*, принадлежащую Security Reference Monitor.

Еще одно преимущество AuthZ заключается в том, что приложения могут указывать AuthZ кэшировать результаты проверок прав доступа для ускорения последующих проверок, где используются те же контекст клиента и дескриптор защиты.

AuthZ полностью документирован в Platform SDK.

Права и привилегии учетных записей

Многие операции, выполняемые процессами, нельзя авторизовать через подсистему защиты доступа к объектам, так как при этом не происходит взаимодействия с конкретным объектом. Например, возможность обходить проверки прав доступа при открытии файлов для резервного копирования является атрибутом учетной записи, а не конкретного объекта. Windows использует как привилегии, так и права учетных записей, чтобы системный администратор мог управлять тем, каким учетным записям разрешено выполнять операции, затрагивающие безопасность.

Привилегия (privilege) — это право (right) учетной записи на выполнение определенной операции, затрагивающей безопасность, например на выключение компьютера или изменение системного времени. Право учетной записи разрешает или запрещает конкретный тип входа в систему, скажем, локальный или интерактивный.

Системный администратор назначает привилегии группам и учетным записям с помощью таких инструментов, как MMC-оснастка Active Directory Users and Groups (Active Directory — пользователи и группы) или редактора локальной политики безопасности (Local Security Policy Editor)*. Запустить этот редактор можно из папки Administrative Tools (Администрирование). На рис. 8-6 показана конфигурация User Rights Assignment (Назначение прав пользователя) редактора локальной политики безопасности, при которой в правой части окна выводится полный список привилегий и прав (учетных записей), доступных в Windows Server 2003. Заметьте, что этот редактор не различает привилегии и права учетных записей. Но вы можете сделать это сами, поскольку любое право, в названии которого встречается слово «logon» («вход»), на самом деле является привилегией.

* В русской версии Windows XP окно этого редактора называется «Локальные параметры безопасности». — *Прим. перев.*

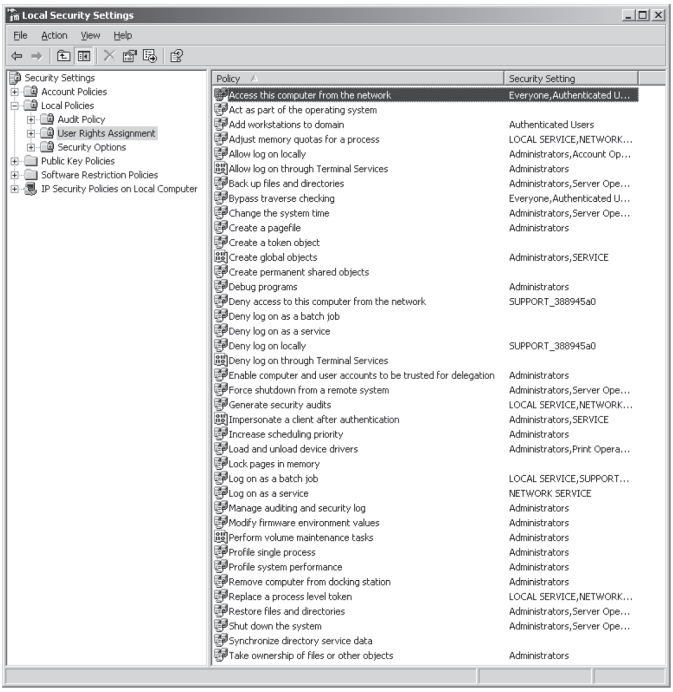


Рис. 8-6. *Конфигурация User Rights Assignment редактора локальной политики безопасности*

Права учетной записи

Права учетной записи не вводятся в действие монитором состояния защиты (Security Reference Monitor, SRM) и не хранятся в маркерах. За вход отвечает функция *LsaLogonUser*. В частности, WinLogon вызывает API-функцию *LogonUser*, когда пользователь интерактивно входит в систему, а *LogonUser* обращается к *LsaLogonUser*. Эта функция принимает параметр, указывающий тип выполняемого входа, который может быть интерактивным, сетевым, пакетным, сервисным, через службу терминала или для разблокировки (unlock).

В ответ на запросы входа служба локальной безопасности (Local Security Authority, LSA) извлекает назначенные пользователю права учетной записи из своей базы данных; эта операция выполняется при попытке пользователя войти в систему. LSA сверяет тип входа с правами учетной записи и по результатам этой проверки отклоняет попытку входа, если у учетной записи нет права, разрешающего данный тип входа, или, напротив, есть право, которое запрещает данный тип входа. Права пользователей, определенные в Windows, перечислены в таблице 8-4.

Windows-приложения могут добавлять или удалять права из учетной записи пользователя через функции *LsaAddAccountRights* и *LsaRemoveAccountRights* соответственно, а также определять, какие права назначены учетной записи, вызывая функцию *LsaEnumerateAccountRights*.

Таблица 8-4. *Права учетной записи*

Право	Описание
Deny logon interactively (Отклонить интерактивный вход), Allow logon interactively (Интерактивный вход)	Используется для интерактивного входа, запрос на который исходит с локального компьютера
Deny logon over the network (Отказ в доступе к компьютеру из сети), Allow logon over the network (Доступ к компьютеру из сети)	Используется для входа, запрос на который исходит с удаленного компьютера
Deny logon through Terminal Services[1] (Запретить вход в систему через службу терминалов), Allow logon through Terminal Services[1] (Разрешать вход в систему через службу терминалов)	Используется для входа через клиент службы терминалов
Deny logon as a service (Отказать во входе в качестве службы), Allow logon as a service (Вход в качестве службы)	Применяется SCM при запуске сервиса под учетной записью определенного пользователя
Deny logon as a batch job (Отказ во входе в качестве пакетного задания), Allow logon as a batch job (Вход в качестве пакетного задания)	Используется при пакетном входе

[1] Впервые введено в Windows XP.

Привилегии

Число привилегий, определяемых в операционной системе, со временем выросло. В отличие от прав пользователей, которые вводятся в действие в одном месте службой LSA, разные привилегии определяются разными компонентами и ими же применяются. Скажем, привилегия отладки, позволяющая процессу обходить проверки прав доступа при открытии описателя другого процесса через API-функцию *OpenProcess*, проверяется диспетчером процессов. Полный список привилегий приведен в таблице 8-5.

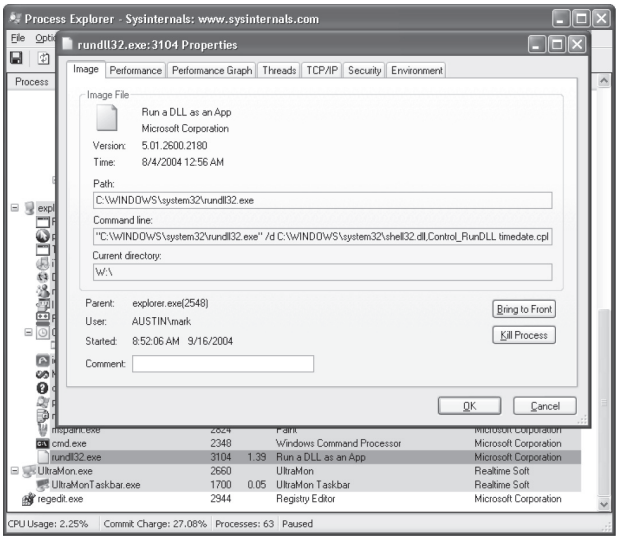
Компонент, которому нужно проверить маркер на наличие некоей привилегии, обращается к API-функции *PrivilegeCheck* или *LsaEnumerateAccountRights*, если он выполняется в пользовательском режиме, либо к *SeSinglePrivilegeCheck* или *SePrivilegeCheck*, если он работает в режиме ядра. API-функции, работающие с привилегиями, ничего не знают о правах учетных записей, но API-функциям, оперирующим с правами, привилегии известны.

В отличие от прав учетной записи привилегии можно включать и отключать. Чтобы проверка привилегии прошла успешно, эта привилегия должна находиться в указанном маркере и должна быть включена. Смысл такой схемы в том, что привилегии должны включаться только при реальном их использовании, и в том, чтобы процесс не мог случайно выполнить привилегированную операцию.

ЭКСПЕРИМЕНТ: наблюдение за включением привилегии

Следующая процедура позволит увидеть, как апплет Date and Time (Дата и время) из Control Panel включает привилегию SeSystemTimePrivilege, исходя из того, что его интерфейс будет использован для изменения даты или времени на компьютере.

1. Войдите в систему под учетной записью, имеющей право «Change the system time» (Изменение системного времени); такая учетная запись обычно входит в группу администраторов или пользователей с правами администраторов.
2. Запустите Process Explorer и выберите для частоты обновления значение Paused.
3. Откройте вкладку Security в окне свойств какого-либо процесса, например Explorer. Вы должны увидеть, что привилегия SeChangeSystemTimePrivilege отключена.
4. Запустите апплет Date and Time из Control Panel и обновите окно Process Explorer. В списке появится новый процесс Rundll, выделенный зеленым цветом.
5. Откройте окно свойств для процесса Rundll (дважды щелкнув имя этого процесса) и убедитесь, что командная строка содержит текст «Timedate.Cpl». Наличие этого аргумента сообщает Rundll (хост-процессу Control Panel DLL) загрузить DLL, реализующую UI, который позволяет изменять дату и время.



6. Перейдите на вкладку Security в окне свойств процесса Rundll и вы увидите, что привилегия SeSystemTimePrivilege включена.

см. след. стр.

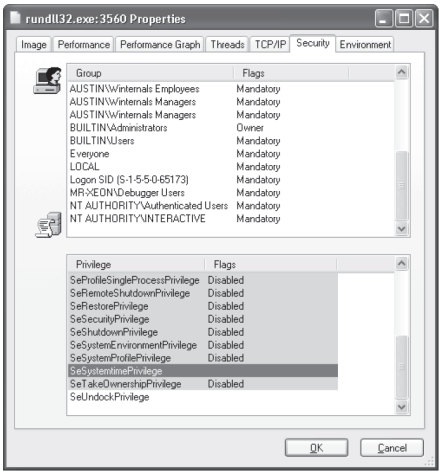


Таблица 8-5. *Привилегии*

Внутреннее название привилегии	Название привилегии в UI*	Описание
SeAssignPrimary-TokenPrivilege	Замена маркера уровня процесса	Проверяется различными компонентами, например функцией *NtSetInformationJob*, которая задает маркер процесса
SeAuditPrivilege	Аудит безопасности	Нужна для генерации событий в журнале событий безопасности с помощью API-функции *ReportEvent*
SeBackupPrivilege	Резервное копирование файлов и каталогов	Заставляет NTFS разрешать следующие виды доступа к любому файлу или каталогу независимо от дескриптора защиты: READ_CONTROL, ACCESS_SYSTEM_SECURITY, FILE_GENERIC_READ, FILE_TRAVERSE. Заметьте: при открытии файла для резервного копирования вызвавший код должен указывать флаг FILE_FLAG_BACKUP_SEMANTICS
SeChangeNotify-Privilege	Обход промежуточных проверок	Используется NTFS во избежание проверки разрешений для промежуточных каталогов при проходе многоуровневых каталогов. Также используется файловыми системами, когда приложения регистрируются на уведомление об изменениях в структуре файловой системы
SeCreateGlobal-Privilege[1]	Создание глобальных объектов	Требуется процессу, чтобы создать объекты «раздел» и «символьная ссылка» в каталогах пространства имен диспетчера объектов, которые закреплены за другим сеансом

* В этом столбце дается перевод английских названий привилегий, а не названия привилегий в русской версии Windows. — *Прим. перев.*

Таблица 8-5. (*продолжение*)

Внутреннее название привилегии	Название привилегии в UI	Описание
SeCreatePagefile-Privilege	Создание страничного файла	Проверяется функцией *NtCreatePaging-File*, применяемой для создания нового страничного файла
SeCreatePerma-nentPrivilege	Создание постоянных разделяемых объектов	Проверяется диспетчером объектов при создании постоянного объекта (не удаляемого в отсутствие ссылок на него)
SeCreateToken-Privilege	Создание объекта «маркер»	Проверяется функцией *NtCreateToken*, создающей объект «маркер»
SeDebug	Отладка программ	При наличии такой привилегии диспетчер процессов разрешает доступ к любому процессу через *NtOpenProcess* независимо от его дескриптора защиты
SeEnableDelega-tionPrivilege[2]	Сделать учетные записи компьютера и пользователя доверяемыми для делегирования	Используется сервисами Active Directory для делегирования аутентифицированных удостоверений
SeImpersonate-Privilege[1]	Подмена клиента после аутентификации	Проверяется диспетчером процессов, когда какому-то потоку нужно использовать маркер для олицетворения, представляющий другого пользователя
SeIncreaseBase-PriorityPrivilege	Повышение приоритета в планировании	Проверяется диспетчером процессов и необходима для повышения приоритета процесса
SeIncreaseQuota-Privilege	Увеличение квот (в Windows 2000) или изменение квот на память для процесса (в Windows XP и Windows Server 2003)	Включается при изменении квоты реестра (только в Windows 2000), пороговых размеров рабочего набора процесса и квот на пулы подкачиваемой и не подкачиваемой памяти для процесса
SeLoadDriver-Privilege	Загрузка и выгрузка драйверов устройств	Проверяется функциями *NtLoadDriver* и *NtUnloadDriver*
SeLockMemory-Privilege	Блокирование страниц в памяти	Проверяется *NtLockVirtualMemory* — реализацией *VirtualLock* в ядре
SeMachineAcco-untPrivilege	Добавление рабочих станций в домен	Проверяется диспетчером учетных записей безопасности (Security Accounts Manager, SAM) на контроллере домена при создании учетной записи компьютера в домене
SeManageVolume-Privilege[2]	Выполнение сервисных операций для тома	Включается драйверами файловой системы при операции открытия тома, необходимой для проверки диска и дефрагментации
SeProfileSingle-ProcessPrivilege	Профилирование одного процесса	Проверяется функцией средства предвыборки (prefetcher), которая возвращает информацию для индивидуального процесса

см. след. стр.

Таблица 8-5. (*продолжение*)

Внутреннее название привилегии	Название привилегии в UI	Описание
SeRemoteShutdownPrivilege	Принудительное выключение с удаленного компьютера	Проверяется Winlogon для удаленного кода, вызывающего функцию *InitiateSystemShutdown*
SeRestorePrivilege	Восстановление файлов и каталогов	Заставляет NTFS разрешать следующие виды доступа к любому файлу или каталогу независимо от дескриптора защиты: WRITE_DAC, WRITE_OWNER, ACCESS_SYSTEM_SECURITY, FILE_GENERIC_WRITE, FILE_ADD_FILE, FILE_ADD_SUBDIRECTORY, DELETE. Заметьте: при открытии файла для восстановления вызвавший код должен указывать флаг FILE_FLAG_BACKUP_SEMANTICS
SeSecurityPrivilege	Управление аудитом и журналом событий безопасности	Требуется для доступа к SACL дескриптора защиты, для чтения и очистки журнала событий безопасности
SeShutdownPrivilege	Выключение системы	Проверяется функциями *NtShutdownSystem* и *NtRaiseHardError*, которые выводят в интерактивной консоли диалоговое окно о возникновении системной ошибки
SeSyncAgentPrivilege	Синхронизация данных службы каталогов	Требуется для использования сервисов синхронизации каталогов LDAP и позволяет владельцу этой привилегии читать все объекты и свойства в каталоге независимо от атрибутов защиты этих объектов и свойств
SeSystemEnvironmentPrivilege	Модификация переменных окружения микрокода	Требуется *NtSetSystemEnvironmentValue* и *NtQuerySystemEnvironmentValue* для модификации и чтения переменных окружения микрокода (firmware environment variables) через HAL
SeSystemProfilePrivilege	Профилирование производительности системы	Проверяется функцией *NtCreateProfile* при профилировании системы. Используется, например, утилитой Kernprof
SeSystemtimePrivilege	Изменение системного времени	Требуется для изменения времени или даты
SeTakeOwnership	Смена владельца файлов и других объектов	Требуется для смены владельца объекта без получения разрешения на избирательный доступ
SeTcbPrivilege	Работа в режиме операционной системы	Проверяется SRM, если в маркере задан идентификатор сеанса, диспетчером Plug and Play при событиях, связанных с созданием и управлением, версией *LogonUser*, реализованной в Windows 2000, *BroadcastSystemMessageEx* при вызове с флагом BSM_ALLDESKTOPS и функцией *LsaRegisterLogonProcess*

Таблица 8-5. (*окончание*)

Внутреннее название привилегии	Название привилегии в UI	Описание
SeUndockPrivilege	Извлечение компьютера из стыковочного узла	Проверяется диспетчером Plug and Play пользовательского режима, когда инициируется извлечение компьютера из стыковочного узла (docking station) или выдается запрос на извлечение устройства

[1] Введено в Windows Server 2003.

[2] Введено в Windows XP.

ЭКСПЕРИМЕНТ: привилегия Bypass Traverse Checking

Если вы являетесь системным администратором, то должны знать о привилегии Bypass Traverse Checking (Обход перекрестной проверки)* (ее внутреннее название — SeNotifyPrivilege) и о том, какие последствия влечет за собой ее включение. Этот эксперимент демонстрирует, что непонимание ее поведения может привести к серьезному нарушению безопасности.

1. Создайте папку, а в ней — новый текстовый файл с каким-нибудь текстом.
2. Перейдите в Explorer к новому файлу и откройте вкладку Security (Безопасность) в его окне свойств. Щелкните кнопку Advanced (Дополнительно) и сбросьте флажок, управляющий наследованием. Выберите Copy (Копировать), когда появится запрос с предложением удалить или скопировать унаследованные разрешения.
3. Далее сделайте так, чтобы по вашей учетной записи нельзя было получить доступ к этой новой папке. Для этого выберите свою учетную запись и в списке разрешений выберите все флажки типа Deny (отклонить или запретить).
4. Запустите Notepad и попробуйте через его UI перейти в новую папку. Вы не сможете этого сделать.
5. В поле File Name (Имя файла) диалогового окна Open (Открыть) введите полный путь к новому файлу. Файл должен открыться.

Если в вашей учетной записи нет привилегии Bypass Traverse Checking, NTFS будет проверять права доступа к каждому каталогу в пути к файлу, когда вы попытаетесь открыть этот файл. И только в таком случае вам будет отказано в доступе к данному файлу.

* Так эта привилегия называется в русской версии Windows XP, но на самом деле никакой перекрестной проверки нет — проверяются промежуточные каталоги в пути к файлу. Поэтому такую привилегию следовало бы назвать «Обход промежуточных проверок». — *Прим. перев.*

Суперпривилегии

Несколько привилегий дают настолько широкие права, что пользователя, которому они назначаются, называют «суперпользователем» — он получает полный контроль над компьютером. Эти привилегии позволяют получать неавторизованный доступ к закрытым ресурсам и выполнять любые операции. Но мы уделим основное внимание применению привилегии на запуск кода, который выдает привилегии, изначально не назначавшиеся пользователю, и при этом не будем забывать, что это может быть использовано для выполнения любой операции на локальном компьютере. В этом разделе перечисляются такие привилегии и рассматриваются способы их применения. Прочие привилегии вроде Lock Pages In Physical Memory (Закрепление страниц в памяти) можно использовать для атак типа «отказ в обслуживании», но мы не станем их обсуждать.

- **Debug programs (Отладка программ)** Пользователь с этой привилегией может открыть любой процесс в системе независимо от его дескриптора защиты. Например, располагая такой привилегией, можно запустить свою программу, которая открывает процесс LSASS, копирует в ее адресное пространство исполняемый код, а затем внедряет поток с помощью API-функции CreateRemoteThread для выполнения внедренного кода в более привилегированном контексте защиты. Этот код мог бы выдавать пользователю дополнительные привилегии и расширять его членство в группах.
- **Take ownership (Смена владельца)*** Эта привилегия позволяет ее обладателю сменить владельца любого защищаемого объекта, просто вписав свой SID в поле владельца в дескрипторе защиты объекта. Вспомните, что владелец всегда получает разрешение на чтение и модификацию DACL дескриптора защиты, поэтому процесс с такой привилегией мог бы изменить DACL, чтобы разрешить себе полный доступ к объекту, а затем закрыть объект и вновь открыть его с правами полного доступа. Это позволило бы увидеть любые конфиденциальные данные и даже подменить системные файлы, выполняемые при обычных системных операциях, например LSASS, своими программами, которые расширяют привилегии некоего пользователя.
- **Restore files and directories (Восстановление файлов и каталогов)** Пользователь с такой привилегией может заменить любой файл в системе на свой — так же, как было описано в предыдущем абзаце.
- **Load and unload device drivers (Загрузка и выгрузка драйверов устройств)** Злоумышленник мог бы воспользоваться этой привилегией для загрузки драйвера устройства в систему. Такие драйверы считаются доверяемыми частями операционной системы, которые выполняются под системной учетной записью, поэтому драйвер мог бы запускать привилегированные программы, назначающие пользователю-злоумышленнику другие права.

---

* В русской версии Windows XP эта привилегия называется «Овладение файлами или иными объектами». — *Прим. перев.*

- **Create a token object (Создание маркерного объекта)** Эта привилегия позволяет создавать объекты «маркеры», представляющие произвольные учетные записи с членством в любых группах и любыми разрешениями.
- **Act as part of operating system (Работа в режиме операционной системы)** Эта привилегия проверяется функцией *LsaRegisterLogonProcess*, вызываемой процессом для установления доверяемого соединения с LSASS. Злоумышленник с такой привилегией может установить доверяемое соединение с LSASS, а затем вызвать *LsaLogonUser* — функцию, используемую для создания новых сеансов входа. *LsaLogonUser* требует указания действительных имени и пароля пользователя и принимает необязательный список SID, добавляемый к начальному маркеру, который создается для нового сеанса входа. В итоге можно было бы использовать свои имя и пароль для создания нового сеанса входа, в маркер которого включены SID более привилегированных групп или пользователей.

Заметьте, что расширенные привилегии не распространяются за границы локальной системы в сети, потому что любое взаимодействие с другим компьютером требует аутентификации контроллером домена и применения доменных паролей. А доменные пароли не хранятся на компьютерах (даже в зашифрованном виде) и поэтому недоступны злонамеренному коду.

Аудит безопасности

События аудита может генерировать диспетчер объектов в результате проверки прав доступа. Их могут генерировать и непосредственно Windows-функции, доступные пользовательским приложениям. Это же право, разумеется, есть и у кода режима ядра. С аудитом связаны две привилегии: SeSecurityPrivilege и SeAuditPrivilege. Для управления журналом событий безопасности, а также для просмотра и изменения SACL объектов процесс должен обладать привилегией SeSecurityPrivilege. Однако процесс, вызывающий системные сервисы аудита, должен обладать привилегией SeAuditPrivilege, чтобы успешно сгенерировать запись аудита в этом журнале.

Решения об аудите конкретного типа событий безопасности принимаются в соответствии с политикой аудита локальной системы. Политика аудита, также называемая локальной политикой безопасности (local security policy), является частью политики безопасности, поддерживаемой LSASS в локальной системе, и настраивается с помощью редактора локальной политики безопасности (рис. 8-7).

При инициализации системы и изменении политики LSASS посылает SRM сообщения, информирующие его о текущей политике аудита. LSASS отвечает за прием записей аудита, генерируемых на основе событий аудита от SRM, их редактирование и передачу Event Logger (регистратору событий). Эти записи посылает именно LSASS (а не SRM), так как он добавляет в них сопутствующие подробности, например информацию, нужную для более полной идентификации процесса, по отношению к которому проводится аудит.

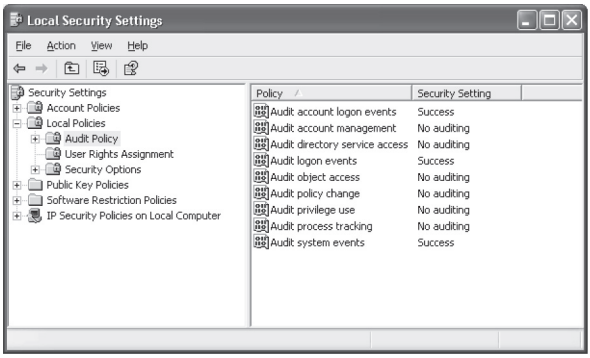


Рис. 8-7. *Конфигурация Audit Policy редактора локальной политики безопасности*

SRM посылает записи аудита LSASS через свое LPC-соединение. После этого Event Logger заносит записи в журнал безопасности. В дополнение к записям аудита, передаваемым SRM, LSASS и SAM тоже генерируют записи аудита, которые LSASS пересылает непосредственно Event Logger; кроме того, AuthZ API позволяет приложениям генерировать записи аудита, определенные этими приложениями. Вся эта схема представлена на рис. 8-8.

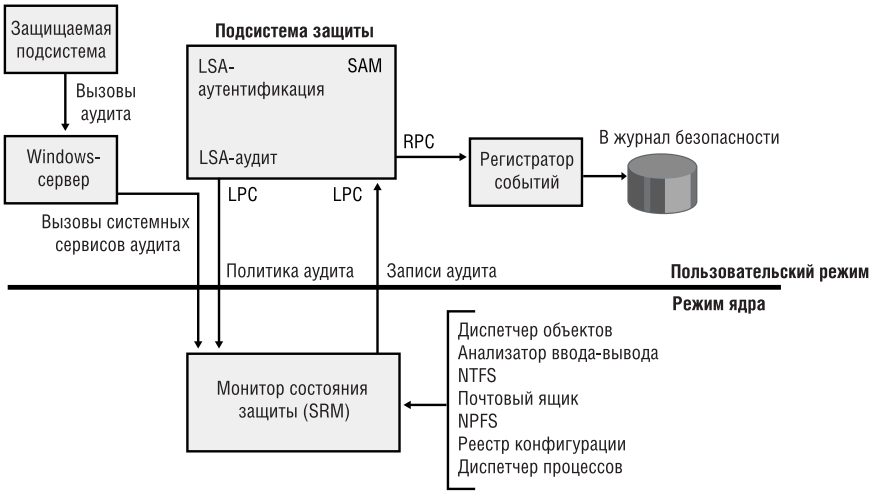


Рис. 8-8. *Схема передачи записей аудита безопасности*

Записи аудита, подлежащие пересылке LSA, помещаются в очередь по мере получения — они не передаются пакетами. Пересылка этих записей осуществляется одним из двух способов. Если запись аудита невелика (меньше максимального размера LPC-сообщения), она посылается как LPC-сообщение. Записи аудита копируются из адресного пространства SRM в адресное пространство процесса Lsass. Если запись аудита велика, SRM делает ее доступной Lsass через разделяемую память и передает Lsass указатель на нее, используя для этого LPC-сообщение.

Рис. 8-9 обобщает изложенные в этой главе концепции, иллюстрируя базовые структуры защиты процессов и потоков. Обратите внимание на то, что у объектов «процесс» и «поток» имеются ACL, равно как и у самих объектов «маркер доступа». Кроме того, на этой иллюстрации показано, что у потоков 2 и 3 есть маркер олицетворения, тогда как поток 1 по умолчанию использует маркер доступа своего процесса.

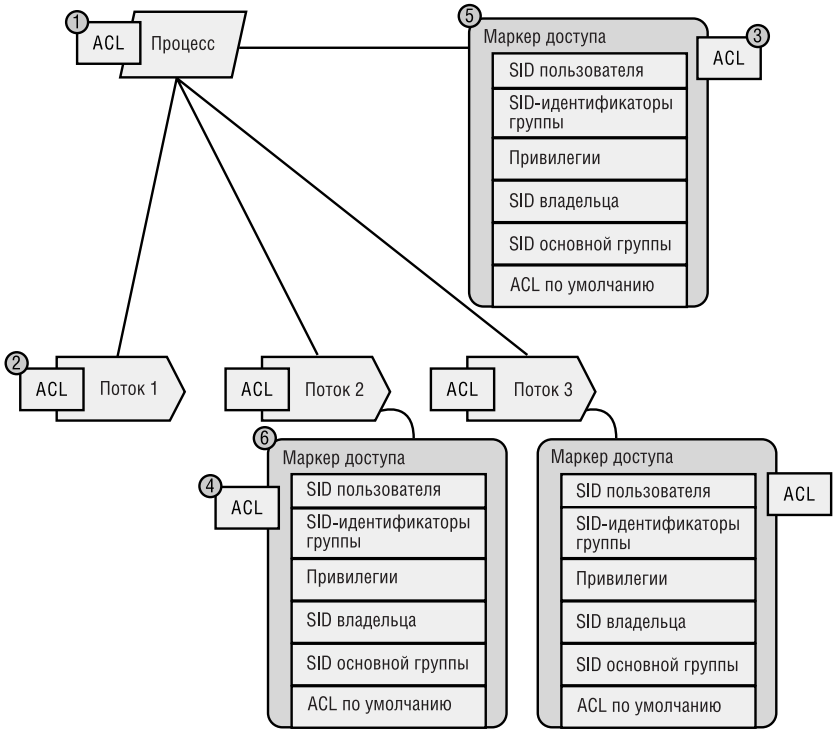


Рис. 8-9. *Структуры защиты процессов и потоков*

Вход в систему

При интерактивном входе в систему (в отличие от входа через сеть) происходит взаимодействие с процессами Winlogon, Lsass, одним или несколькими пакетами аутентификации, а также SAM или Active Directory. *Пакеты аутентификации* (authentication packages) — это DLL-модули, выполняющие проверки, связанные с аутентификацией. Пакетом аутентификации Windows для интерактивного входа в домен является Kerberos, а MSV1_0 — аналогичным пакетом для интерактивного входа на локальные компьютеры, доменного входа в доверяемые домены под управлением версий Windows, предшествовавших Windows 2000, а также для входа в отсутствие контроллера домена.

Winlogon — доверяемый процесс, отвечающий за управление взаимодействием с пользователем в связи с защитой. Он координирует вход, запускает первый процесс при входе в систему данного пользователя, обрабатывает выход из системы и управляет множеством других операций, имеющих отношение к защите, — вводом паролей при регистрации, сменой паролей, блокированием и разблокированием рабочих станций и т. д. Процесс Winlogon должен обеспечить невидимость операций, связанных с защитой, другим активным процессам. Так, Winlogon гарантирует, что в ходе этих операций недоверяемый процесс не сможет перехватить управление рабочим столом и таким образом получить доступ к паролю.

Winlogon получает имя и пароль пользователя через Graphical Identification and Authentication (GINA) DLL. Стандартная GINA — \Windows\System32\Msgina.dll. Msgina выводит диалоговое окно для входа в систему. Позволяя заменять Msgina другими GINA-библиотеками, Windows дает возможность менять механизмы идентификации пользователей. Например, сторонний разработчик может создать GINA для поддержки устройства распознавания отпечатков пальцев и для выборки паролей пользователей из зашифрованной базы данных.

Winlogon — единственный процесс, который перехватывает запросы на регистрацию с клавиатуры. Получив имя и пароль пользователя от GINA, Winlogon вызывает LSASS для аутентификации этого пользователя. Если аутентификация прошла успешно, процесс Winlogon активизирует оболочку. Схема взаимодействия между компонентами, участвующими в процессе регистрации, показана на рис. 8-10.

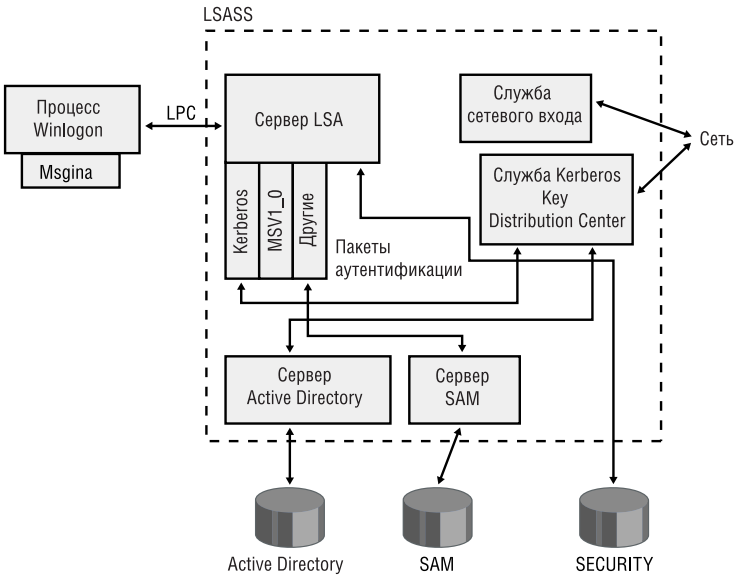


Рис. 8-10. *Компоненты, участвующие в процессе входа*

Winlogon не только поддерживает альтернативные GINA, но и может загружать дополнительные DLL провайдеров доступа к сетям, необходимые для вторичной аутентификации. Это позволяет сразу нескольким сетевым провайдерам получать идентификационные и регистрационные данные в процессе обычного входа пользователя в систему. Входя в систему под управлением Windows, пользователь может одновременно аутентифицироваться и на UNIX-сервере. После этого он получит доступ к ресурсам UNIX-сервера с компьютера под управлением Windows без дополнительной аутентификации. Эта функциональность является одной из форм *унифицированной регистрации* (single sign-on).

Инициализация Winlogon

При инициализации системы, когда ни одно пользовательское приложение еще не активно, Winlogon выполняет ряд операций, обеспечивающих ему контроль над рабочей станцией с момента готовности системы к взаимодействию с пользователем.

1. Создает и открывает интерактивный объект WindowStation (например, \Windows\WindowStations\WinSta0 в пространстве имен диспетчера объектов), представляющий клавиатуру, мышь и монитор. Далее создает дескриптор защиты станции с одним ACE, содержащим только системный SID. Этот уникальный дескриптор безопасности гарантирует, что другой процесс получит доступ к рабочей станции, только если Winlogon явно разрешит это.
2. Создает и открывает два объекта «рабочий стол»: для приложений (\Windows\WinSta0\Default, также известный как интерактивный рабочий стол) и Winlogon (\Windows\WinSta0\Winlogon, также известный как защищенный рабочий стол). Защита объекта «рабочий стол» Winlogon организуется так, чтобы к нему мог обращаться только Winlogon. Другой объект «рабочий стол» доступен как Winlogon, так и пользователям. Следовательно, пока активен объект «рабочий стол» Winlogon, никакой другой процесс не получает доступа к коду и данным, сопоставленным с этим рабочим столом. Эта функциональность используется Windows для защиты операций, требующих передачи паролей, а также для блокировки и разблокировки рабочего стола.
3. До входа какого-либо пользователя в систему видимым рабочим столом является объект «рабочий стол» Winlogon. После входа нажатие клавиш Ctrl+Alt+Del вызывает переключение объектов «рабочий стол» — с Default на Winlogon. (Это объясняет, почему после нажатия Ctrl+Alt+Del с рабочего стола исчезают все окна и почему они возвращаются, как только закрывается диалоговое окно Windows Security.) Таким образом, SAS всегда активизирует защищенный рабочий стол, контролируемый Winlogon.
4. Устанавливает LPC-соединение с LSASS через *LsaAuthenticationPort* (вызовом *LsaRegisterLogonProcess*). Это соединение понадобится для обмена

информацией при входе и выходе пользователя из системы и при операциях с паролем.

Далее Winlogon настраивает оконную среду.

5. Инициализирует и регистрирует структуру данных оконного класса, которая сопоставляет процедуру Winlogon с создаваемым ею окном.
6. Регистрирует SAS, сопоставляя ее с только что созданным окном. Это гарантирует, что ввод пользователем SAS будет вызывать именно оконную процедуру Winlogon и что программы типа троянских коней не смогут перехватывать управление при вводе SAS.
7. Регистрирует окно, чтобы при выходе пользователя вызывалась процедура, сопоставленная с этим окном. Подсистема Windows проверяет, что запросивший уведомление процесс является именно Winlogon.

Как реализована SAS

SAS безопасна потому, что никакое приложение не может перехватить комбинацию клавиш Ctrl+Alt+Del или воспрепятствовать его получение Winlogon. Winlogon использует документированную API-функцию *RegisterHotKey* для резервирования комбинации клавиш Ctrl+Alt+Del, поэтому подсистема ввода Windows, обнаружив эту комбинацию, посылает специальное сообщение окну, создаваемому Winlogon для приема таких уведомлений. Любая зарезервированная комбинация клавиш посылается только тому процессу, который зарезервировал ее, и лишь поток, зарезервировавший данную комбинацию клавиш, может отменить ее регистрацию (через API-функцию *UnregisterHotKey*), так что троянская программа не в состоянии забрать на себя SAS.

Windows-функция *SetWindowsHook* позволяет приложению установить процедуру-ловушку, вызываемую при каждом нажатии клавиш еще до обработки какой-либо комбинации, и модифицировать эти клавиши. Однако в коде обработки комбинаций клавиш содержится специальный блок case для Ctrl+Alt+Del, который отключает ловушки, исключая возможность перехвата этой последовательности. Кроме того, если интерактивный рабочий стол блокирован, обрабатываются только комбинации клавиш, принадлежащие Winlogon.

Как только при инициализации системы создается рабочий стол Winlogon, он становится активным рабочим столом. Причем активный рабочий стол Winlogon всегда заблокирован. Winlogon разблокирует свой рабочий стол лишь для переключения на рабочий стол приложений или экранной заставки. (Блокировать или разблокировать рабочий стол может только процесс Winlogon.)

Этапы входа пользователя

Регистрация начинается, когда пользователь нажимает комбинацию клавиш SAS (по умолчанию — Ctrl+Alt+Del). После этого Winlogon вызывает GINA,

чтобы получить имя и пароль пользователя. Winlogon также создает уникальный локальный SID для этого пользователя и назначает его данному экземпляру объекта «рабочий стол» (который представляет клавиатуру, экран и мышь). Winlogon передает этот SID в LSASS при вызове *LsaLogonUser*. Если вход пользователя прошел успешно, этот SID будет включен в маркер процесса входа (logon process token) — такой шаг предпринимается для защиты доступа к объекту «рабочий стол». Например, второй вход по той же учетной записи, но в другой системе, не предоставит доступа для записи к объекту «рабочий стол» первого компьютера, так как в его маркере не будет SID, полученного при втором входе.

После ввода имени и пароля пользователя Winlogon получает описатель пакета аутентификации вызовом Lsass-функции *LsaLookupAuthenticationPackage*. Эти пакеты перечисляются в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\Lsa. Winlogon передает пакету данные входа через *LsaLogonUser*. После того как пакет аутентифицирует пользователя, Winlogon продолжает процесс входа этого пользователя. Если ни один из пакетов не сообщает об успешной аутентификации, процесс входа прекращается.

Windows использует два стандартных пакета аутентификации при интерактивном входе: Kerberos и MSV1_0. Пакет аутентификации по умолчанию в автономной системе Windows — MSV1_0 (\Windows\System32\Msv1_0.dll); он реализует протокол LAN Manager 2. LSASS также использует MSV1_0 на компьютерах, входящих в домен, чтобы аутентифицировать домены и компьютеры под управлением версий Windows до Windows 2000, не способные найти контроллер домена для аутентификации. (Отключенные от сети портативные компьютеры относятся к той же категории.) Пакет аутентификации Kerberos (\Windows\System32\Kerberos.dll) используется на компьютерах, входящих в домены Windows. Этот пакет во взаимодействии со службами Kerberos, выполняемыми на контроллере домена, поддерживает протокол Kerberos версии 5 (ревизии 6). Данный протокол определен в RFC 1510. (Подробнее о стандарте Kerberos см. на сайте Internet Engineering Task Force по ссылке *www.ietf.org*.)

Пакет аутентификации MSV1_0 принимает имя пользователя и хэшированную версию пароля и посылает локальному SAM запрос на получение информации из учетной записи, включая пароль, группы, в которые входит пользователь, и список ограничений по данной учетной записи. Сначала MSV1_0 проверяет ограничения, например разрешенное время или типы доступа. Если ограничения из базы данных SAM запрещают регистрацию пользователя в это время суток, MSV1_0 возвращает LSA статус отказа.

Далее MSV1_0 сравнивает хэшированный пароль и имя пользователя с теми, которые хранятся в SAM. В случае кэшированного доменного входа MSV1_0 обращается к кэшированной информации через функции LSASS, отвечающие за сохранение и получение «секретов» из базы данных LSA (куст реестра SECURITY). Если эти данные совпадают, MSV1_0 генерирует LUID сеанса входа и создает собственно сеанс входа вызовом LSASS. При этом MSV1_0 сопоставляет данный уникальный идентификатор с сеансом и пе-

редает данные, необходимые для того, чтобы в конечном счете создать маркер доступа для пользователя. (Вспомните, что маркер доступа включает SID пользователя, SID групп и назначенные привилегии.)

ПРИМЕЧАНИЕ MSV1_0 не кэширует весь хэш пароля пользователя в реестре, так как это позволило бы любому лицу, имеющему физический доступ к системе, легко скомпрометировать доменную учетную запись пользователя и получить доступ к зашифрованным файлам и к сетевым ресурсам, к которым данный пользователь имеет право обращаться. Поэтому MSV1_0 кэширует лишь половину хэша. Этой половины достаточно для проверки правильности пароля пользователя, но недостаточно для получения доступа к ключам EFS и для аутентификации в домене вместо этого пользователя, так как эти операции требуют полного хэша.

Если MSV1_0 нужно аутентифицировать пользователя с удаленной системы, например при его регистрации в доверяемом домене под управлением версий Windows до Windows 2000, то MSV1_0 взаимодействует с экземпляром Netlogon в удаленной системе через службу Netlogon (сетевого входа в систему). Netlogon в удаленной системе взаимодействует с пакетом аутентификации MSV1_0 этой системы, передавая результаты аутентификации системе, в которой выполняется вход.

Базовая последовательность действий при аутентификации Kerberos в основном та же, что и в случае MSV1_0. Однако в большинстве случаев доменный вход проходит на рабочих станциях или серверах, включенных в домен (а не на контроллере домена), поэтому пакет в процессе аутентификации должен взаимодействовать с ними через сеть. Взаимодействие этого пакета со службой Kerberos на контроллере домена осуществляется через TCP/IP-порт Kerberos (88). Служба Kerberos Key Distribution Center (\Windows\System32\Kdcsvc.dll), реализующая протокол аутентификации Kerberos, выполняется в процессе Lsass на контроллерах домена.

После проверки хэшированной информации об имени и пароле пользователя с помощью объектов учетных записей пользователей (user account objects) Active Directory (через сервер Active Directory, \Windows\System32\Ntdsa.dll) Kdcsvc возвращает доменные удостоверения LSASS, который при успешном входе передает через сеть результат аутентификации и удостоверения пользователя той системе, где выполняется вход.

ПРИМЕЧАНИЕ Приведенное здесь описание аутентификации пакетом Kerberos сильно упрощено, и тем не менее оно иллюстрирует роль различных компонентов в этом процессе. Хотя протокол аутентификации Kerberos играет ключевую роль в обеспечении распределенной защиты доменов в Windows, его детальное рассмотрение выходит за рамки нашей книги.

Как только учетные данные аутентифицированы, LSASS ищет в базе данных локальной политики разрешенный пользователю тип доступа — интер-

активный, сетевой, пакетный или сервисный. Если тип запрошенного входа в систему не соответствует разрешенному, вход прекращается. LSASS удаляет только что созданный сеанс входа, освобождая его структуры данных, и сообщает Winlogon о неудаче. Winlogon в свою очередь сообщает об этом пользователю. Если же запрошенный тип входа в систему разрешается, LSASS добавляет любые дополнительные идентификаторы защиты (например, Everyone, Interactive и т. п.). Затем он проверяет в своей базе данных привилегии, назначенные всем идентификаторам данного пользователя, и включает эти привилегии в маркер доступа пользователя.

Собрав всю необходимую информацию, LSASS вызывает исполнительную систему для создания маркера доступа. Исполнительная система создает основной маркер доступа для интерактивного или сервисного сеанса и маркер олицетворения для сетевого сеанса. После успешного создания маркера доступа LSASS дублирует его, создавая описатель, который может быть передан Winlogon, а свой описатель закрывает. Если нужно, проводится аудит операции входа. На этом этапе LSASS сообщает Winlogon об успешном входе и возвращает описатель маркера доступа, LUID сеанса входа и информацию из профиля, полученную от пакета аутентификации (если она есть).

ЭКСПЕРИМЕНТ: перечисление активных сеансов входа

Пока существует хотя бы один маркер с данным LUID сеанса входа, Windows считает этот сеанс активным. С помощью утилиты LogonSessions (*www.sysinternals.com*), которая использует функцию *LsaEnumerateLogonSessions* (документированную в Platform SDK), можно перечислить активные сеансы входа:

```
C:\>logonsessions
LogonSessions 1.0
Copyright (C) 2004 Bryce Cogswell
Sysinternals - www.sysinternals.com

[0] Logon session 00000000:000003e7:
    User name:    AUSTIN\MARKLAP$
    Auth package: NTLM
    Logon type:   (none)
    Session:      0
    Sid:          S-1-5-18
    Logon time:   5/13/2004 9:05:43 AM
    Logon server:
    DNS Domain:
    UPN:
[1] Logon session 00000000:0000d37d:
    User name:
    Auth package: NTLM
    Logon type:   (none)
    Session:      0
    Sid:          (none)
```

см. след. стр.

```
    Logon time:   8/17/2004 5:08:10 PM
    Logon server:
    DNS Domain:
    UPN:
[2] Logon session 00000000:000003e4:
    User name:    NT AUTHORITY\NETWORK SERVICE
    Auth package: Negotiate
    Logon type:   Service
    Session:      0
    Sid:          S-1-5-20
    Logon time:   8/17/2004 5:08:18 PM
    Logon server:
    DNS Domain:
    UPN:
...

[8] Logon session 00000000:0079da73:
    User name:    MARKLAP\Administrator
    Auth package: MICROSOFT_AUTHENTICATION_PACKAGE_V1_0
    Logon type:   NetworkCleartext
    Session:      0
    Sid:          S-1-5-21-1787744166-3910675280-2727264193-500
    Logon time:   5/13/2004 12:14:20 PM
    Logon server: MARKLAP
    DNS Domain:
    UPN:
```

В информацию, сообщаемую по каждому сеансу, включаются SID и имя пользователя, сопоставленные с данным сеансом, а также пакет аутентификации и время входа. Заметьте, что пакет аутентификации Negotiate, отмеченный в сеансе входа 2, выполняет аутентификацию через Kerberos или NTLM в зависимости от того, какой из них больше подходит для данного запроса на аутентификацию.

LUID для сеанса показывается в строке «Logon Session» блока информации по каждому сеансу, и с помощью утилиты Handle (также доступной с *www.sysinternals.com*) можно найти маркеры, представляющие конкретный сеанс входа. Например, чтобы найти маркеры для сеанса входа 8 в предыдущем примере вывода, вы могли бы ввести такую команду:

```
C:\>handle -a 79da73
  43c: Token         MARKLAP\Administrator:79da73
```

Далее Winlogon просматривает параметр реестра HKLM\SOFTWARE\Microsoft\Windows NT\Current Version\Winlogon\Userinit и создает процесс для запуска программ, указанных в строковом значении этого параметра (там могут присутствовать имена нескольких EXE-файлов, разделенные запятыми). Значение этого параметра по умолчанию приводит к запуску Useri-

nit.exe, который загружает профиль пользователя, а затем создает процесс для запуска программ, перечисленных в HKCU\SOFTWARE\Microsoft\Windows NT\Current Version\Winlogon\Shell, если такой параметр есть. Если же этого параметра нет, Userinit.exe обращается к параметру HKLM\SOFTWARE\Microsoft\Windows NT\Current Version\Winlogon\Shell, который по умолчанию задает Explorer.exe. После этого Userinit завершается — вот почему Process Explorer показывает Explorer.exe как процесс, не имеющий предка. Подробнее о том, что происходит в процессе входа, см. в главе 5.

Политики ограниченного использования программ

Злонамеренный код вроде вирусов и червей создает все больше проблем. В Windows XP введен механизм Software Restriction Policies (Политики ограниченного использования программ), который позволяет администраторам контролировать образы и сценарии, выполняемые в их системах. Узел Software Restriction Policies в редакторе локальной политики безопасности (рис. 8-11) служит интерфейсом управления для политик выполнения кода на компьютере, хотя возможны и политики, индивидуальные для пользователей; в последнем случае применяются доменные политики групп.

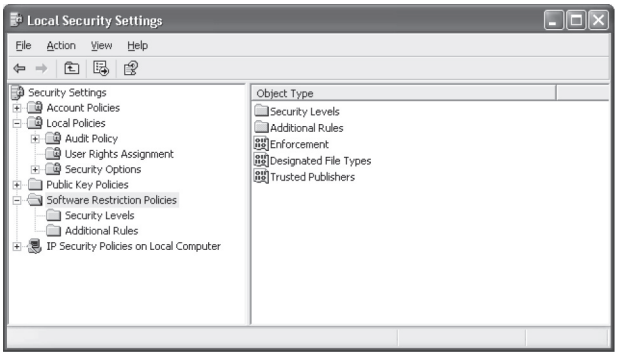


Рис. 8-11. *Конфигурация Software Restriction Policy редактора локальной политики безопасности*

Узел Software Restriction Policies (Политики ограниченного использования программ) содержит несколько глобальных параметров.

- Параметр Enforcement (Принудительный) определяет, как применяются политики ограничения — к библиотекам вроде DLL, только к пользователям или к пользователям и администраторам.
- Параметр Designated File Types (Назначенные типы файлов) регистрирует расширения файлов, которые считаются исполняемыми.
- Параметр Trusted Publishers (Доверенные издатели) контролирует, кто имеет право решать, каким издателям сертификатов можно доверять.

При настройке параметра для конкретного сценария или образа администратор может указать системе распознавать этот сценарий или образ по его пути, хэшу, зоне Интернета (как определено в Internet Explorer) или по криптографическому сертификату, а также сопоставить его с уровнем безопасности Disallowed (Не разрешено) либо Unrestricted (Неограниченный).

Политики ограниченного использования программ применяются внутри различных компонентов, где файлы рассматриваются как содержащие исполняемый код. Некоторые из таких компонентов перечислены ниже.

- Windows-функция CreateProcess (\Windows\System32\Kernel32.dll) пользовательского режима применяет эти политики к исполняемым образам.
- Код загрузки DLL в Ntdll (\Windows\System32\Ntdll.dll) применяет эти политики к DLL.
- Командная оболочка Windows (\Windows\System32\Cmd.exe) применяет эти политики к командным файлам.
- Компоненты Windows Scripting Host, запускающие сценарии, — \Windows\System32\Cscript.exe (для сценариев командной строки), \Windows\System32\Wscript.exe (для UI-сценариев) и \Windows\System32\Scrobj.dll (для объектов-сценариев) — применяют эти политики к сценариям.

Каждый из этих компонентов определяет, действуют ли политики ограничения, по значению параметра реестра HKLM\SOFTWARE\Policies\Microsoft\Windows\Safer\CodeIdentifiers\TransparentEnabled. Если он равен 1, то политики действуют. Далее каждый из компонентов проверяет, подпадает ли код, который он собирается выполнить, под действие одного из правил, указанных в подразделе раздела CodeIdentifiers, и, если да, следует ли разрешить выполнение. Если ни одно из правил к данному коду не относится, его выполнение зависит от политики по умолчанию, определяемой параметром DefaultLevel в разделе CodeIdentifiers.

Software Restriction Policies — мощное средство для предотвращения запуска неавторизованного кода и сценариев, но только при правильном применении. Если политика по умолчанию не запрещает выполнение, то в образ, который не разрешено запускать в данной системе, можно внести минимальные изменения, и это позволит обойти правило и запустить данный образ.

ЭКСПЕРИМЕНТ: наблюдение за применением политики ограниченного использования программ

Вы можете косвенно убедиться в применении политик ограниченного использования программ, наблюдая за обращениями к реестру при попытке выполнения образа, запуск которого запрещен.

1. Запустите secpol.msc, чтобы открыть редактор локальной политики безопасности, и перейдите в узел Software Restriction Policies (Политики ограниченного использования программ).
2. Выберите Create New Policies (Создать новые политики) из контекстного меню, если такие политики не определены.

3. Создайте правило, запрещающее путь \Windows\System32\Notepad.exe.
4. Запустите Regmon и установите включающий фильтр для «Safer» (описание Regmon см. в главе 4).
5. Откройте окно командной строки и попробуйте запустить Notepad.

Ваша попытка запуска Notepad должна закончиться появлением сообщения о том, что вам запрещен запуск указанной программы, и Regmon должна показать, как командная оболочка (cmd.exe) запрашивает политики ограничения на локальном компьютере.

Резюме

Windows поддерживает большой набор функций защиты, соответствующий ключевым требованиям как правительственных организаций, так и коммерческих структур. В этой главе мы кратко рассмотрели внутренние компоненты, лежащие в основе функций защиты.

В следующей главе мы обсудим последний из основных компонентов исполнительной системы, описываемых в этой книге, — подсистему ввода-вывода.

ГЛАВА 9

Подсистема ввода-вывода

Подсистема ввода-вывода в Microsoft Windows состоит из нескольких компонентов исполнительной системы, которые совместно управляют аппаратными устройствами и предоставляют интерфейсы для обращения к ним системе и приложениям. В этой главе мы сначала перечислим цели разработки подсистемы ввода-вывода, повлиявшие на ее реализацию. Затем мы рассмотрим ее компоненты, в том числе диспетчер ввода-вывода, диспетчер Plug and Play (PnP) и диспетчер электропитания. Далее исследуем структуру подсистемы ввода-вывода и различные типы драйверов устройств. Мы также обсудим основные структуры данных, описывающие устройства, драйверы устройств и запросы на ввод-вывод, а потом перейдем к этапу обработки запросов на ввод-вывод. В завершение будет рассказано о том, как распознаются устройства, как устанавливаются их драйверы и как осуществляется управление электропитанием.

Компоненты подсистемы ввода-вывода

Согласно целям, поставленным при разработке, подсистема ввода-вывода в Windows должна обеспечивать приложениям абстракцию устройств — как аппаратных (физических), так и программных (виртуальных или логических) — и при этом предоставлять следующую функциональность:

- стандартные средства безопасности и именования устройств для защиты разделяемых ресурсов (описание модели защиты см. в главе 8);
- высокопроизводительный асинхронный пакетный ввод-вывод для поддержки масштабируемых приложений;
- сервисы для написания драйверов устройств на высокоуровневом языке и упрощения их переноса между разными аппаратными платформами;
- поддержку многоуровневой модели и расширяемости для добавления драйверов, модифицирующих поведение других драйверов или устройств без внесения изменений в них;
- динамическую загрузку и выгрузку драйверов устройств, чтобы драйверы можно было загружать по требованию и не расходовать системные ресурсы без необходимости;
- поддержку Plug and Play, благодаря которой система находит и устанавливает драйверы для нового оборудования, а затем выделяет им нужные аппаратные ресурсы;

- управление электропитанием, чтобы система и отдельные устройства могли переходить в состояния с низким энергопотреблением;
- поддержку множества устанавливаемых файловых систем, в том числе FAT, CDFS (файловую систему CD-ROM), UDF (Universal Disk Format) и NTFS (подробнее о типах и архитектуре файловых систем см. в главе 12).
- поддержку Windows Management Instrumentation (WMI) и средств диагностики, позволяющую управлять драйверами и вести мониторинг за ними через WMI-приложения и сценарии. (Описание WMI см. в главе 4.)

Для реализации этой функциональности подсистема ввода-вывода в Windows состоит из нескольких компонентов исполнительной системы и драйверов устройств (рис. 9-1).

- Центральное место в этой подсистеме занимает диспетчер ввода-вывода; он подключает приложения и системные компоненты к виртуальным, логическим и физическим устройствам, а также определяет инфраструктуру, поддерживающую драйверы устройств.
- Драйвер устройства, как правило, предоставляет интерфейс ввода-вывода для устройств конкретного типа. Такие драйверы принимают от диспетчера ввода-вывода команды, предназначенные управляемым ими устройствам, и уведомляют диспетчер ввода-вывода о выполнении этих команд. Драйверы часто используют этот диспетчер для пересылки команд ввода-вывода другим драйверам, задействованным в реализации интерфейса того же устройства и участвующим в управлении им.
- Диспетчер PnP работает в тесном взаимодействии с диспетчером ввода-вывода и *драйвером шины* (bus driver) — одной из разновидностей драйверов устройств. Он управляет выделением аппаратных ресурсов, а также распознает устройства и реагирует на их подключение или отключение. Диспетчер PnP и драйверы шины отвечают за загрузку соответствующего драйвера при обнаружении нового устройства. Если устройство добавляется в систему, в которой нет нужного драйвера устройства, компоненты исполнительной системы, отвечающие за поддержку PnP, вызывают сервисы установки устройств, поддерживаемые диспетчером PnP пользовательского режима.
- Диспетчер электропитания, также в тесном взаимодействии с диспетчером ввода-вывода, управляет системой и драйверами устройств при их переходе в различные состояния энергопотребления.
- Процедуры поддержки Windows Management Instrumentation (WMI) (Инструментарий управления Windows), образующие провайдер WDM (Windows Driver Model) WMI, позволяют драйверам устройств выступать в роли провайдеров, взаимодействуя со службой WMI пользовательского режима через провайдер WDM WMI. (Подробнее о WMI см. раздел «Windows Management Instrumentation» главы 4.)
- Реестр служит в качестве базы данных, в которой хранится описание основных устройств, подключенных к системе, а также параметры инициализации драйверов и конфигурационные настройки (см. главу 4).

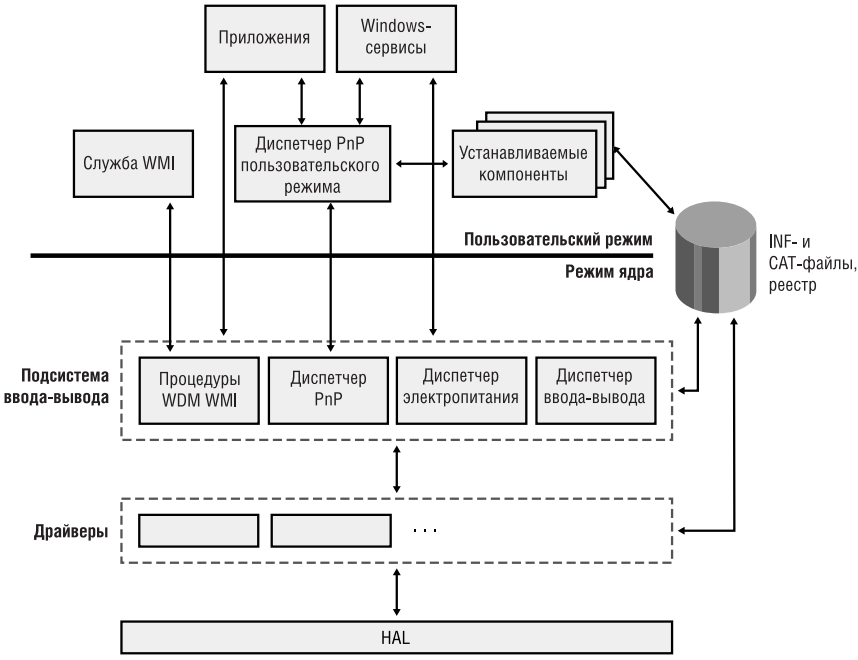


Рис. 9-1. *Компоненты подсистемы ввода-вывода*

- Для установки драйверов используются INF-файлы; они связывают конкретное аппаратное устройство с драйвером, который берет на себя ведущую роль в управлении этим устройством. Содержимое INF-файла состоит из инструкций, описывающих соответствующее устройство, исходное и целевое местонахождение файлов драйвера, изменения, которые нужно внести в реестр при установке драйвера, и информацию о зависимостях драйвера. В CAT-файлах хранятся цифровые подписи, которые удостоверяют файлы драйверов, прошедших испытания в лаборатории Microsoft Windows Hardware Quality Lab (WHQL).
- Уровень абстрагирования от оборудования (HAL) изолирует драйверы от специфических особенностей конкретных процессоров и контроллеров прерываний, поддерживая API, скрывающие межплатформенные различия. В сущности HAL является драйвером шины для тех устройств на материнской плате компьютера, которые не контролируются другими драйверами.

Диспетчер ввода-вывода

Диспетчер ввода-вывода (I/O manager) определяет модель доставки запросов на ввод-вывод драйверам устройств. Подсистема ввода-вывода управляется пакетами. Большинство запросов ввода-вывода представляется *пакетами запросов ввода-вывода* (I/O request packets, IRP), передаваемых от одного компонента подсистемы ввода-вывода другому. (Как вы еще убедитесь, исключением является быстрый ввод-вывод, при котором IRP не использу-

ются.) Подсистема ввода-вывода позволяет индивидуальному потоку приложения управлять сразу несколькими запросами на ввод-вывод. IRP — это структура данных, которая содержит информацию, полностью описывающую запрос ввода-вывода (подробнее об IRP см. раздел «Пакеты запросов ввода-вывода» далее в этой главе).

Диспетчер ввода-вывода создает IRP (представляющий операцию ввода-вывода), передает указатель на IRP соответствующему драйверу и удаляет пакет по завершении операции ввода-вывода. Драйвер, получивший IRP, выполняет указанную в пакете операцию и возвращает IRP диспетчеру ввода-вывода, чтобы тот либо завершил эту операцию, либо передал пакет другому драйверу для дальнейшей обработки.

Диспетчер ввода-вывода не только создает и уничтожает IRP, но и содержит общий для различных драйверов код, который они используют при обработке ввода-вывода. Благодаря этому драйверы стали проще и компактнее. Так, одна из функций диспетчера ввода-вывода позволяет драйверу вызывать другие драйверы. Этот диспетчер также управляет буферами запросов ввода-вывода, таймаутами драйверов и регистрирует, какие устанавливаемые файловые системы загружаются в операционную систему. Драйверы устройств могут вызывать около сотни функций, предоставляемых диспетчером ввода-вывода.

Диспетчер ввода-вывода также предоставляет гибкие сервисы ввода-вывода, на основе которых подсистемы окружения (например, Windows и POSIX) реализуют свои функции. В их число входят весьма изощренные сервисы асинхронного ввода-вывода, которые дают возможность разработчикам создавать высокопроизводительные масштабируемые серверные приложения.

Унифицированный модульный интерфейс драйверов позволяет диспетчеру ввода-вывода вызывать любой драйвер, ничего не зная о его структуре и внутреннем устройстве. Операционная система обрабатывает запросы на ввод-вывод так, будто они адресованы файлам; драйвер преобразует запросы к виртуальному файлу в запросы, специфичные для устройства. Драйверы также могут вызывать друг друга (через диспетчер ввода-вывода), обеспечивая многоуровневую независимую обработку запросов на ввод-вывод.

Кроме обычных функций для открытия, закрытия, чтения и записи подсистема ввода-вывода Windows предоставляет ряд дополнительных функций, например для асинхронного, прямого и буферизованного ввода-вывода, а также для ввода-вывода по механизму «scatter/gather»* (см. раздел «Типы ввода-вывода» далее в этой главе).

Типичная обработка ввода-вывода

Большинство операций ввода-вывода не требует участия всех компонентов подсистемы ввода-вывода. Как правило, запрос на ввод-вывод выдается при-

* Механизм, позволяющий интерпретировать, записывать и считывать физически нелинейную область памяти как единое целое. — *Прим. перев.*

ложением, выполняющим соответствующую операцию (например, чтение данных с устройства); такие операции обрабатываются диспетчером ввода-вывода, одним или несколькими драйверами устройств и HAL.

Как уже упоминалось, в Windows потоки выполняют операции ввода-вывода над виртуальными файлами. Операционная система абстрагирует все запросы на ввод-вывод, скрывая тот факт, что конечное устройство ввода-вывода может и не быть устройством с файловой структурой. Это позволяет обобщить интерфейс между приложениями и устройствами. Таким образом, виртуальный файл относится к любому источнику или приемнику ввода-вывода (файлу, каталогу, именованному каналу и почтовому ящику), который рассматривается как файл. Все считываемые или записываемые данные представляются простыми потоками байтов, направляемыми в виртуальные файлы. Приложения пользовательского режима (к какой бы подсистеме они ни относились — Windows или POSIX) вызывают документированные функции, которые в свою очередь обращаются к внутренним функциям подсистемы ввода-вывода для чтения/записи файла и для выполнения других операций. Запросы, адресованные виртуальным файлам, диспетчер ввода-вывода динамически направляет соответствующим драйверам устройств. Базовая схема обработки запроса на ввод-вывод показана на рис. 9-2.

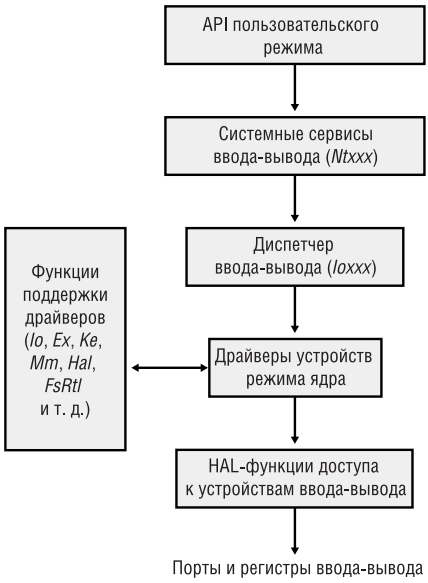


Рис. 9-2. *Схема обработки типичного запроса на ввод-вывод*

Далее мы детальнее рассмотрим эти компоненты, исследуем различные типы драйверов устройств, рассмотрим их структуру, загрузку, инициализацию и обработку запросов на ввод-вывод. Кроме того, мы обсудим роль и функциональность диспетчеров PnP и электропитания.

Драйверы устройств

Для интеграции с диспетчером ввода-вывода и другими компонентами подсистемы ввода-вывода драйвер устройства должен быть написан в соответствии с правилами, специфичными для управляемого им типа устройств и для его роли в управлении такими устройствами. Здесь мы познакомимся с типами драйверов устройств, поддерживаемых Windows, и исследуем внутреннюю структуру драйвера устройства.

Типы драйверов устройств

Windows поддерживает множество типов драйверов устройств и сред их программирования. Среды программирования могут различаться даже для драйверов одного типа — в зависимости от типа устройства, для которого предназначен драйвер. Драйверы могут работать в двух режимах: в пользовательском или в режиме ядра. Windows поддерживает несколько типов драйверов пользовательского режима:

- **Драйверы виртуальных устройств (virtual device drivers, VDD)** Используются для эмуляции 16-разрядных программ MS-DOS. Они перехватывают обращения таких программ к портам ввода-вывода и транслируют их в вызовы Windows-функций ввода-вывода, передаваемые реальным драйверам устройств. Поскольку Windows является полностью защищенной операционной системой, программы MS-DOS пользовательского режима не могут напрямую обращаться к аппаратным средствам — они должны делать это через драйверы устройств режима ядра.
- **Драйверы принтеров** Драйверы подсистемы Windows, которые транслируют аппаратно-независимые запросы на графические операции в команды, специфичные для принтера. Далее эти команды обычно направляются драйверу режима ядра, например драйверу параллельного порта (Parport.sys) или драйверу порта принтера на USB-шине (Usb-print.sys).

  В этой главе основное внимание уделяется драйверам устройств, работающим в режиме ядра. Эти драйверы можно разбить на несколько основных категорий.
- **Драйверы файловой системы** Принимают запросы на ввод-вывод и выполняют их, выдавая более специфические запросы драйверам устройств массовой памяти или сетевым драйверам.
- **PnP-драйверы** Драйверы, работающие с оборудованием и интегрируемые с диспетчерами электропитания и PnP. В их число входят драйверы для устройств массовой памяти, видеоадаптеров, устройств ввода и сетевых адаптеров.
- **Драйверы, не отвечающие спецификации Plug and Play** Также называются расширениями ядра. Расширяют функциональность системы, предоставляя доступ из пользовательского режима к сервисам и драйверам режима ядра. Они не интегрируются с диспетчерами PnP и электро-

питания. К ним, в частности, относятся драйверы протоколов и сетевого API. Драйвер Regmon, описанный в главе 4, тоже входит в эту категорию.

Категория драйверов режима ядра подразделяется на группы в зависимости от модели, на которой они основаны, и их роли в обслуживании запросов к устройствам.

WDM-драйверы

Это драйверы устройств, отвечающие спецификации Windows Driver Model (WDM). WDM требует от драйверов поддержки управления электропитанием, Plug and Play и WMI. Большинство драйверов Plug and Play построены как раз на модели WDM. Эта модель реализована в Windows, Windows 98 и Windows Millennium Edition, поэтому WDM-драйверы этих операционных систем совместимы на уровне исходного кода, а во многих случаях и на уровне двоичного кода. Существует три типа WDM-драйверов.

- **Драйверы шин** Управляют логическими или физическими шинами. Примеры шин — PCMCIA, PCI, USB, IEEE 1394, ISA. Драйвер шины отвечает за распознавание устройств, подключенных к управляемой им шине, оповещение о них диспетчера PnP и управление параметрами электропитания шины.
- **Функциональные драйверы** Управляют конкретным типом устройств. Драйверы шин представляют устройства функциональным драйверам через диспетчер PnP. Функциональным считается драйвер, экспортирующий рабочий интерфейс устройства операционной системе. Как правило, это драйвер, больше других знающий о функционировании определенного устройства.
- **Драйверы фильтров** Занимающие более высокий логический уровень, чем функциональные драйверы, они дополняют функциональность или изменяют поведение устройства либо другого драйвера. Так, утилиту для перехвата ввода с клавиатуры можно реализовать в виде драйвера фильтра клавиатуры, расположенного на более высоком уровне, чем функциональный драйвер клавиатуры.

В WDM ни один драйвер не отвечает за все аспекты управления конкретным устройством. Драйвер шины определяет изменения в составе устройств на шине (при подключении и отключении устройств), помогает диспетчеру PnP в перечислении устройств на шине, обращается к специфичным для шины регистрам и в некоторых случаях управляет электропитанием подключенных к шине устройств. К аппаратной части устройства обычно обращается только функциональный драйвер.

ПРИМЕЧАНИЕ В Windows 2000, Windows XP и Windows Server 2003 уровень HAL играет несколько иную роль, чем в Windows NT. До Windows 2000 сторонним поставщикам оборудования, которым нужно было добавить поддержку аппаратных шин, не поддерживаемых самой операционной системой, приходилось разрабатывать собственный HAL. Windows 2000, Windows XP и Windows Server 2003 позволяют сторонним разработчикам реализовать поддержку таких шин в виде драйверов шин.

Многоуровневые драйверы

Поддержка индивидуального устройства часто распределяется между несколькими драйверами, каждый из которых обеспечивает часть функциональности, необходимой для нормальной работы устройства. Кроме WDM-драйверов шин, функциональных драйверов и драйверов фильтров, оборудование могут поддерживать и следующие компоненты.

- **Драйверы классов устройств (class drivers)** Реализуют обработку ввода-вывода для конкретного класса устройств, например дисковых устройств, ленточных накопителей или приводов CD-ROM, где аппаратные интерфейсы стандартизированы и один драйвер может обслуживать аналогичные устройства от множества производителей.
- **Порт-драйверы (port drivers)** Обрабатывают запросы на ввод-вывод, специфичные для определенного типа порта ввода-вывода, например SCSI. Порт-драйверы реализуются как библиотеки функций режима ядра, а не как драйверы устройств.
- **Минипорт-драйверы (miniport drivers)** Преобразуют универсальные запросы ввода-вывода к порту конкретного типа в запросы, специфичные для адаптера конкретного типа, например для SCSI-адаптера. Минипорт-драйверы являются истинными драйверами устройств, которые импортируют функции, предоставляемые порт-драйвером.

Вот пример, который демонстрирует, как работают драйверы устройств. Драйвер файловой системы принимает запрос на запись данных в определенное место конкретного файла. Он преобразует его в запрос на запись определенного числа байтов по определенному «логическому» адресу на диске. После этого он передает этот запрос (через диспетчер ввода-вывода) простому драйверу диска. Последний в свою очередь преобразует запрос в физический адрес на диске (цилиндр/дорожка/сектор) и позиционирует головки дискового устройства для записи данных. Эта схема действий показана на рис. 9-3.

Эта схема иллюстрирует разделение труда между двумя драйверами. Диспетчер ввода-вывода получает запрос на запись, в котором адрес записи относителен началу конкретного файла. Далее диспетчер ввода-вывода передает запрос драйверу файловой системы, который преобразует информацию запроса в адрес начала записи на диске и число байтов, которые нуж-

но записать на диск. Драйвер файловой системы передает через диспетчер ввода-вывода запрос драйверу диска, который транслирует его в физический адрес на диске и переносит нужные данные.

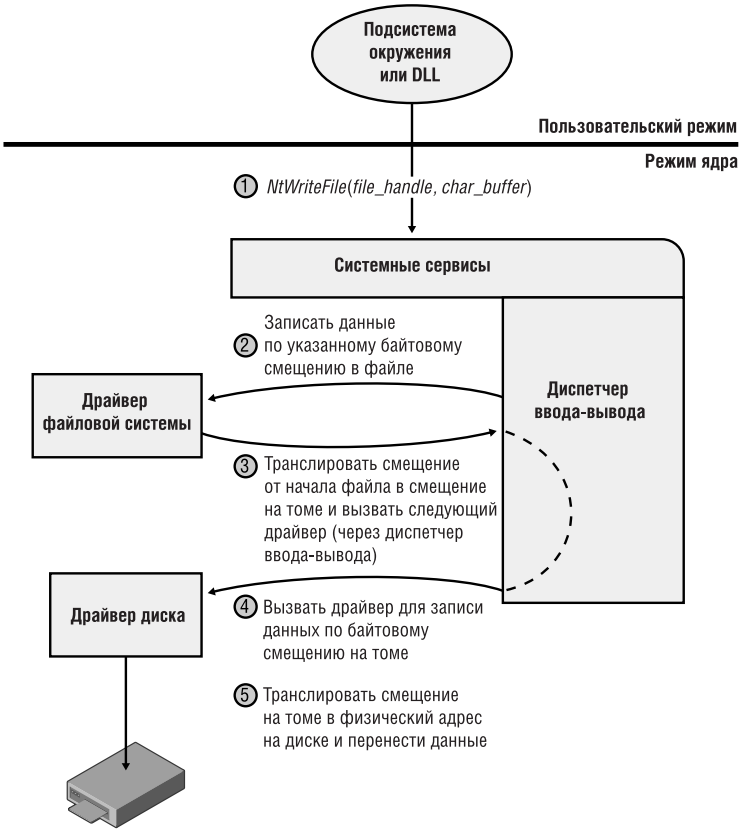


Рис. 9-3. *Взаимодействие драйвера файловой системы и драйвера диска*

Поскольку все драйверы — и устройств, и файловой системы — предоставляют операционной системе одинаковую инфраструктуру, в их иерархию легко добавить еще один драйвер, не изменяя существующие драйверы или подсистему ввода-вывода. Например, введя соответствующий драйвер, можно логически представить несколько дисков как один большой диск. Такой драйвер, кстати, имеется в Windows — он обеспечивает поддержку отказоустойчивых дисков. (Хотя этот драйвер присутствует во всех версиях Windows, поддержка отказоустойчивых дисков доступна лишь в серверных версиях Windows.) Драйвер диспетчера томов вполне логично размещается между драйверами файловой системы и дисков, как показано на рис. 9-4. Подробнее о драйверах диспетчера томов см. в главе 10.

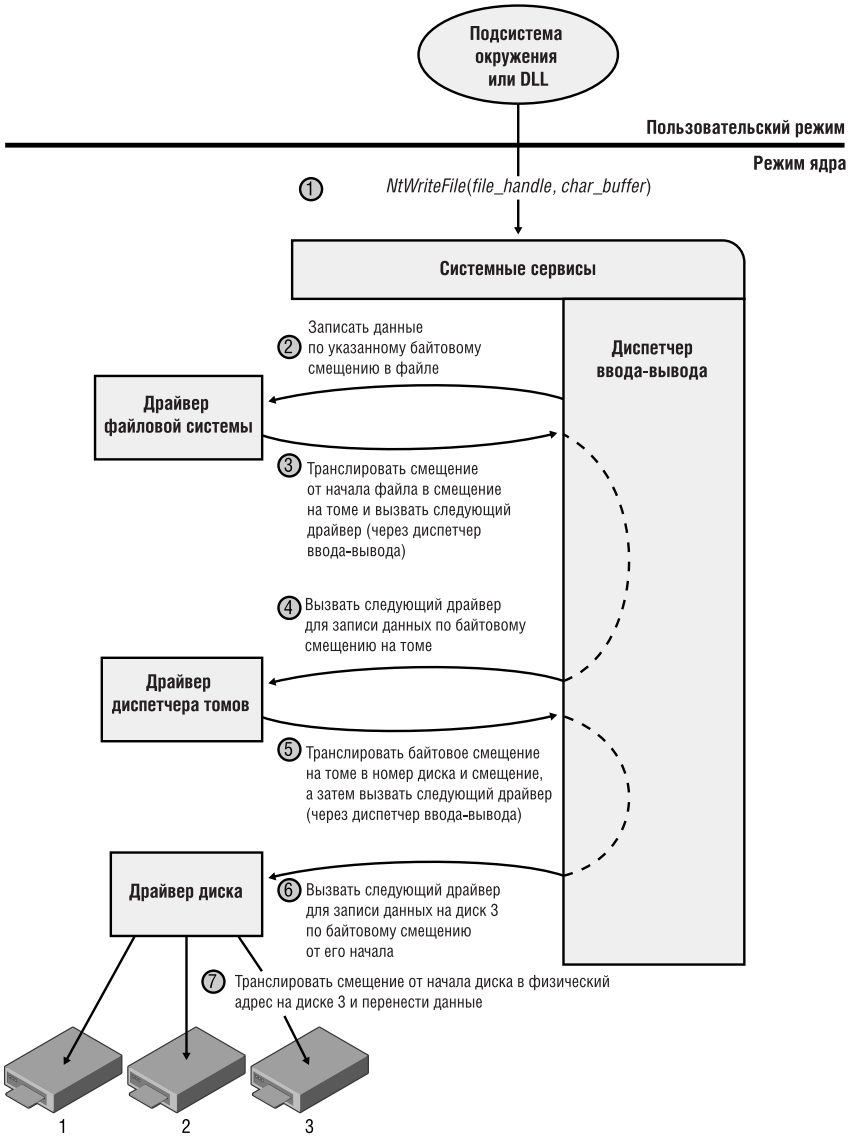
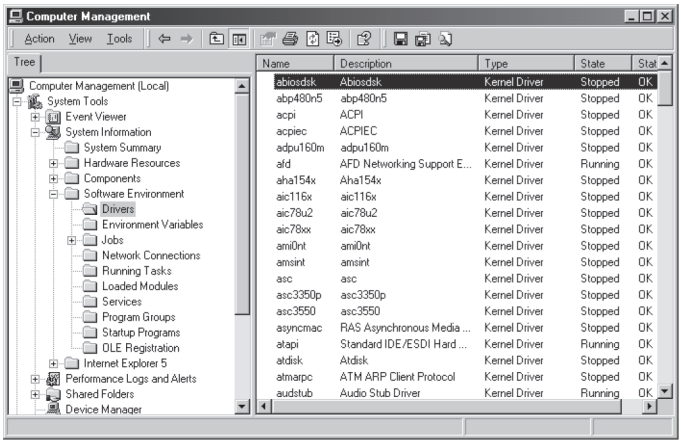


Рис. 9-4. *Добавление промежуточного драйвера*

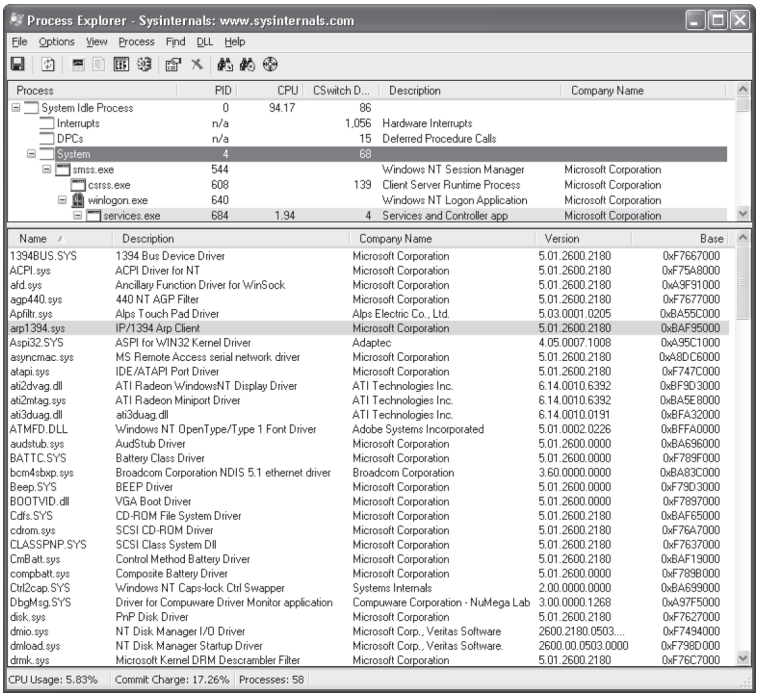
ЭКСПЕРИМЕНТ: просмотр списка загруженных драйверов

Вы можете увидеть список зарегистрированных драйверов в системе Windows 2000, перейдя в раздел Drivers (Драйверы) оснастки Computer Management (Управление компьютером) в Microsoft Management Console (MMC) (Консоль управления Microsoft) или щелкнув правой кнопкой мыши значок My Computer (Мой компьютер) на рабочем столе и выбрав из контекстного меню команду Manage (Управление). Оснастка Computer Management также доступна в подменю Administrative Tools (Администрирование). Чтобы добраться до раздела Drivers в Computer Management, последовательно раскройте узлы System Tools (Служебные программы), System Information (Сведения о системе) и Software Environment (Программная среда), как показано ниже.



В Windows XP и Windows Server 2003 можно получить ту же информацию, запустив утилиту Msinfo32.exe из диалогового окна Run (Запуск программы). Выберите System Drivers (Системные драйверы) в Software Environment (Программная среда) для вывода списка драйверов, сконфигурированных в системе. Те из них, которые загружены на данный момент, помечены словом «Yes» (Да) в столбце Started (Работает).

Список загруженных драйверов режима ядра можно просмотреть и с помощью Process Explorer (*www.sysinternals.com*). Запустите Process Explorer, укажите процесс System и выберите DLLs из подменю Lower Pane в меню View. Process Explorer перечисляет загруженные драйверы, их имена, информацию о версиях, включая название компании и описание, а также адрес загрузки (предполагается, что вы настроили отображение соответствующих столбцов в окне Process Explorer).



Наконец, если вы изучаете аварийный дамп (или «снимок» работающей системы) с помощью отладчика ядра, то можете получить аналогичные сведения командой *lm kv*:

```
kd> lm kv
start    end        module name
804d4000 806aa280   nt           (pdb symbols)
c:\Symbols\ntoskrnl.pdb\
FB1EDACE71FB4812A5D5132819D72E523\ntoskrnl.pdb
    Loaded symbol image file: ntoskrnl.exe
    Image path: ntoskrnl.exe
    Timestamp: Thu Apr 24 10:57:43 2003 (3EA80977)
    Checksum: 001E311B
    ImageSize : 001D6280
    File version:     5.1.2600.1151
    Product version:  5.1.2600.1151
    File flags:       0 (Mask 3F)
    File OS:          40004 NT Windows
    File type:        1.0 App
    File date:        00000000.00000000
    Translations:     0409.04b0
    CompanyName:      Microsoft Corporation
    ProductName:      Microsoft" Windows" Operating System
    InternalName:     ntoskrnl.exe
    OriginalFilename: ntoskrnl.exe
```

см. след. стр.

```
    ProductVersion:   5.1.2600.1151
    FileVersion:      5.1.2600.1151 (xpsp2.030422-1633)
    FileDescription:  NT Kernel & System
    LegalCopyright:    Microsoft Corporation. All rights reserved.
806ab000 806bde80   hal          (deferred)
    Image path: halacpi.dll
    Timestamp: Thu Aug 29 03:05:02 2002 (3D6DD5AE)
    Checksum: 000203BD
    ImageSize : 00012E80
    Translations:     0000.04b0 0000.04e0 0409.04b0 0409.04e0
a8b8e000 a8bb4e80   kmixer       (deferred)
    Image path: \SystemRoot\system32\drivers\kmixer.sys
    Timestamp: Thu Aug 29 03:32:28 2002 (3D6DDC1C)
    Checksum: 00032574
    ImageSize : 00026E80
    Translations:     0000.04b0 0000.04e0 0409.04b0 0409.04e0
```

Структура драйвера

Выполнением драйверов устройств управляет подсистема ввода-вывода. Драйвер устройства состоит из набора процедур, вызываемых на различных этапах обработки запроса ввода-вывода. Основные процедуры драйвера показаны на рис. 9-5.

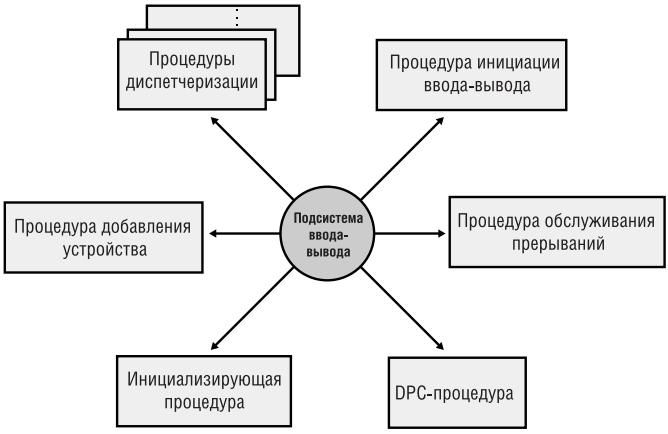


Рис. 9-5. *Основные процедуры драйвера устройства*

- **Инициализирующая процедура** Диспетчер ввода-вывода выполняет инициализирующую процедуру драйвера (которая обычно называется *DriverEntry*) при загрузке этого драйвера в операционную систему. Данная процедура регистрирует остальные процедуры драйвера в диспетчере ввода-вывода, заполняя соответствующей информацией системные

структуры данных, и выполняет необходимую глобальную инициализацию драйвера.

- **Процедура добавления устройства** Такие процедуры реализуются в драйверах, поддерживающих Plug and Play. Через эту процедуру диспетчер PnP посылает драйверу уведомление при обнаружении устройства, за которое отвечает данный драйвер. Выполняя эту процедуру, драйвер обычно создает объект «устройство», представляющий аппаратное устройство.
- **Процедуры диспетчеризации** Это основные функции, предоставляемые драйвером устройства, например для открытия, закрытия, чтения, записи и реализации других возможностей устройства, файловой системы или сети. Диспетчер ввода-вывода, вызванный для выполнения операции ввода-вывода, генерирует IRP и обращается к драйверу через одну из его процедур диспетчеризации.
- **Процедура инициации ввода-вывода** С помощью этой процедуры драйвер может инициировать передачу данных как на устройство, так и с него. Эта процедура определяется лишь в драйверах, использующих поддержку диспетчера ввода-вывода для помещения входящих запросов в очередь. Диспетчер ввода-вывода ставит в очередь IRP для драйвера, гарантируя одновременную обработку им только одного IRP. Большинство драйверов обрабатывают сразу несколько IRP, но создание очереди имеет смысл для некоторых драйверов, в частности для драйвера клавиатуры.
- **Процедура обслуживания прерываний (ISR)** Когда устройство генерирует прерывание, диспетчер прерываний ядра передает управление этой процедуре. В модели ввода-вывода Windows процедуры ISR работают на уровне DIRQL (Device IRQL), поэтому они выполняют минимум действий во избежание слишком продолжительной блокировки прерываний более низкого уровня (подробнее об IRQL см. главу 3). Для выполнения остальной части обработки прерывания ISR ставит в очередь DPC (deferred procedure call), выполняемый при более низком IRQL (уровня «DPC/dispatch»). ISR имеются лишь в драйверах устройств, управляемых прерываниями, — например в драйвере файловой системы ISR нет.
- **DPC-процедура обработки прерываний** DPC-процедура выполняет основную часть обработки прерывания, оставшуюся после выполнения ISR. Она работает при более низком IRQL (уровня «DPC/dispatch»), чем ISR, чтобы не блокировать без необходимости другие прерывания. DPC-процедура инициирует завершение текущей операции ввода-вывода и выполнение следующей операции ввода-вывода из очереди на данном устройстве.

У многих драйверов устройств имеются процедуры, не показанные на рис. 9-5.

- **Одна или несколько процедур завершения ввода-вывода** У драйвера могут быть процедуры завершения ввода-вывода, уведомляющие его об окончании обработки IRP драйвером более низкого уровня. Например, диспетчер ввода-вывода вызывает процедуру завершения ввода-вывода драйве-

ра файловой системы, когда драйвер устройства заканчивает передачу данных в файл или из него. Эта процедура уведомляет драйвер файловой системы об удачном или неудачном завершении операции или о ее отмене, а также позволяет драйверу файловой системы освободить ресурсы.

- **Процедура отмены ввода-вывода** Если операция ввода-вывода может быть отменена, драйвер определяет одну или более процедур отмены ввода-вывода. Получив IRP для запроса ввода-вывода, который может быть отменен, драйвер связывает с IRP процедуру отмены. Если поток, выдавший запрос на ввод-вывод, завершается до окончания обработки запроса или отменяет операцию (например, вызовом Windows-функции *CancelIo*), диспетчер ввода-вывода выполняет процедуру отмены, связанную с IRP (если таковая есть). Процедура отмены отвечает за выполнение любых действий, необходимых для освобождения всех ресурсов, выделенных при обработке IRP, а также за завершение IRP со статусом отмены.
- **Процедура выгрузки** Эта процедура освобождает все системные ресурсы, задействованные драйвером, после чего диспетчер ввода-вывода может удалить их из памяти. При выполнении процедуры выгрузки обычно освобождаются ресурсы, выделенные процедурой инициализации. Драйвер может загружаться и выгружаться во время работы системы.
- **Процедура уведомления о завершении работы системы** Эта процедура позволяет драйверу проводить очистку при завершении работы системы.
- **Процедуры регистрации ошибок** При возникновении неожиданных ошибок (например, когда на диске появляется поврежденный блок), процедуры регистрации ошибок, принадлежащие драйверу, уведомляют о них диспетчер ввода-вывода. Последний записывает эту информацию в файл журнала ошибок.

  ПРИМЕЧАНИЕ Большинство драйверов устройств написано на C. Применение языка ассемблера крайне не рекомендуется из-за его сложности и из-за того, что он затрудняет перенос драйвера между аппаратными архитектурами вроде x86, x64 и IA64.

Объекты «драйвер» и «устройство»

Когда поток открывает описатель объекта «файл» (этот процесс описывается в разделе «Обработка ввода-вывода» далее в этой главе), диспетчер ввода-вывода, исходя из имени этого объекта, должен определить, к какому драйверу (или драйверам) нужно обратиться для обработки запроса. Более того, диспетчер ввода-вывода должен знать, где найти эту информацию, когда в следующий раз поток вновь воспользуется тем же описателем файла. Для этого предназначены следующие объекты.

- Объект «драйвер», представляющий отдельный драйвер в системе. Именно от этого объекта диспетчер ввода-вывода получает адрес процедуры диспетчеризации (точки входа) драйвера.
- Объект «устройство», представляющий физическое или логическое устройство в системе и описывающий его характеристики, например границы выравнивания буферов и адреса очередей для приема IRP, поступающих на это устройство.

Диспетчер ввода-вывода создает объект «драйвер» при загрузке в систему соответствующего драйвера и вызывает его инициализирующую процедуру (например, *DriverEntry*), которая записывает в атрибуты объекта точки входа этого драйвера.

После загрузки драйвер может создавать объекты «устройство» для представления устройств или даже для формирования интерфейса драйвера (вызовом *IoCreateDevice* или *IoCreateDeviceSecure*). Однако большинство PnP-драйверов создают объекты «устройство» с помощью своих процедур добавления устройств, когда диспетчер PnP информирует их о присутствии управляемого ими устройства. С другой стороны, драйверы, не отвечающие спецификации Plug and Play, создают объекты «устройство» при вызове диспетчером ввода-вывода их инициализирующих процедур. Диспетчер ввода-вывода выгружает драйвер после удаления его последнего объекта «устройство», когда ссылок на устройство больше нет.

Создавая объект «устройство», драйвер может присвоить ему имя. Тогда этот объект помещается в пространство имен диспетчера объектов. Драйвер может определить имя этого объекта явно или позволить диспетчеру ввода-вывода сгенерировать его автоматически (о пространстве имен диспетчера объектов см. главу 3). По соглашению объекты «устройство» помещаются в каталог \Device пространства имен, недоступный приложениям через Windows API.

> **ПРИМЕЧАНИЕ** Некоторые драйверы размещают объекты «устройство» в каталогах, отличных от \Device. Так, диспетчер томов Logical Disk Manager создает объекты «устройство», представляющие разделы жесткого диска, в каталоге \Device\HarddiskDmVolumes (подробнее на эту тему см. главу 10).

Чтобы сделать объект «устройство» доступным для приложений, драйвер должен создать в каталоге \Global?? (или в каталоге \?? в Windows 2000) символьную ссылку на имя этого объекта в каталоге \Device. Драйверы, не поддерживающие Plug and Play, и драйверы файловой системы обычно создают символьную ссылку с общеизвестным именем (скажем, \Device\Hardware2). Поскольку общеизвестные имена не срабатывают в средах с динамически меняющимся составом оборудования, PnP-драйверы предоставляют один или несколько интерфейсов через функцию *IoRegisterDeviceInterface*, передавая ей GUID, определяющий тип предоставляемой функциональности. GUID являются 128-битными числами, которые можно генерировать с помощью утилиты Guidgen, входящей в состав DDK и Platform SDK. Диапазон чисел,

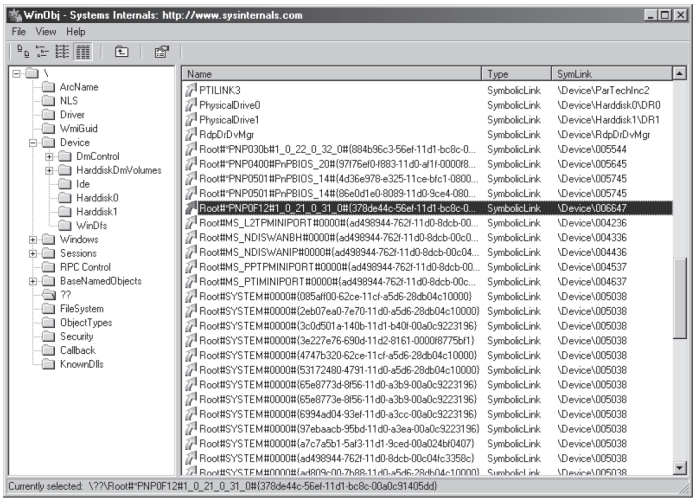
который может быть представлен 128 битами, гарантирует, что каждый GUID, созданный этой утилитой, всегда будет глобально уникальным.

IoRegisterDeviceInterface определяет символьную ссылку, сопоставляемую с экземпляром объекта «устройство». Однако, прежде чем диспетчер ввода-вывода действительно создаст ссылку, драйвер должен вызвать функцию *IoSetDeviceInterfaceState*, чтобы разрешить использование интерфейса этого устройства. Обычно драйвер делает это, когда диспетчер PnP посылает ему команду *start-device* для запуска устройства.

Приложение, которому нужно открыть объект «устройство», представленный GUID-идентификатором, может вызывать PnP-функции настройки, например *SetupDiEnumDeviceInterfaces* для перечисления интерфейсов, доступных по конкретному GUID, и получения имен символьных ссылок, с помощью которых может быть открыт объект «устройство». Чтобы получить дополнительную информацию (например, автоматически сгенерированное имя устройства), приложение вызывает функцию *SetupDiGetDeviceInterfaceDetail* для всех устройств, перечисленных *SetupDiEnumDeviceInterfaces*. Получив от *SetupDiGetDeviceInterfaceDetail* имя устройства, приложение обращается к Windows-функции *CreateFile*, чтобы открыть устройство и получить его описатель.

ЭКСПЕРИМЕНТ: просмотр каталога \Device

Для просмотра имен устройств в каталоге \Device пространства имен диспетчера объектов можно использовать утилиту Winobj (*www.sysinternals.com*) или команду *!object* отладчика ядра. Ниже показан пример символьной ссылки, созданной диспетчером ввода-вывода и указывающей на объект «устройство» с автоматически сгенерированным именем.



Команда *!object* отладчика ядра для каталога \Device выводит следующую информацию.

```
kd> !object \device
Object: e100c4a0  Type: (8a4f3178) Directory
    ObjectHeader: e100c488
    HandleCount: 0  PointerCount: 301
    Directory Object: e10011e8  Name: Device
    65535 symbolic links snapped through this directory

    Hash Address  Type          Name
    ---- -------  ----          ----
     00  8a437398 Device        KsecDD
         8a4a56f0 Device        Ndis
         8a0ed5c0 Device        ProcExp
         8a1ddb40 Device        Beep
         8a336d38 Device        0000008e
         8a4ed730 Device        00000032
         8a4ee4f0 Device        00000025
         8a4b5030 Device        00000019
     01  8a2303e8 Device        Netbios
         8a258030 Device        0000008f
         8a4ed4f0 Device        00000033
         8a4ee2b0 Device        00000026
     02  8a1756d8 Device        KSENUM#00000001
         8a4ec730 Device        00000040
         8a20fd58 Device        Ip
         8a2ef660 Device        RDP_CONSOLE0
         8a4ed2b0 Device        00000034
         8a4b4030 Device        00000027
     03  8a4ec4f0 Device        00000041
         8a15bf18 Device        0000009e
         8a2947d8 Device        {EFF45047-C948-4D32-86B5-736480DDBB9C}
         8a1b38e0 Device        Fips
         8a0b6038 Device        Video0
         8a288b48 Device        RDP_CONSOLE1
         8a2501f8 Device        KeyboardClass0
         8a4b3030 Device        00000035
         8a4b4df0 Device        00000028
     04  8a181030 Device        NDProxy
         8a4ec2b0 Device        00000042
         8a00b038 Device        Video1
         8a189030 Device        KeyboardClass1
    ...
```

При выполнении команды *!object* с указанием объекта-каталога диспетчера объектов отладчик ядра записывает дамп содержимого каталога в соответствии с его внутренней организацией в диспетчере объектов. Для ускорения поиска каталог хранит объекты в хэш-таблице, основанной на хэше имен объектов, поэтому команда *!object* перечисляет объекты так, как они хранятся в каждой корзине (bucket) хэш-таблицы каталога.

Как видно на рис. 9-6, объект «устройство» ссылается на свой объект «драйвер», благодаря чему диспетчер ввода-вывода знает, из какого драйвера нужно вызвать процедуру при получении запроса ввода-вывода. С помощью объекта «устройство» он находит объект «драйвер», который представляет драйвер, обслуживающий устройство. После этого он обращается к объекту «драйвер», используя номер функции из исходного запроса; каждый номер функции соответствует точке входа драйвера (номера функций на рис. 9-6 подробнее описываются в разделе «Блок стека IRP» далее в этой главе).

С объектом «драйвер» нередко сопоставляется несколько объектов «устройство». Список объектов «устройство» представляет физические и логические устройства, управляемые драйвером. Так, для каждого раздела жесткого диска имеется отдельный объект «устройство» с информацией, специфичной для данного раздела. Но для обращения ко всем разделам используется один и тот же драйвер жесткого диска. При выгрузке драйвера из системы диспетчер ввода-вывода с помощью очереди объектов «устройство» определяет устройства, на которые повлияет удаление драйвера.

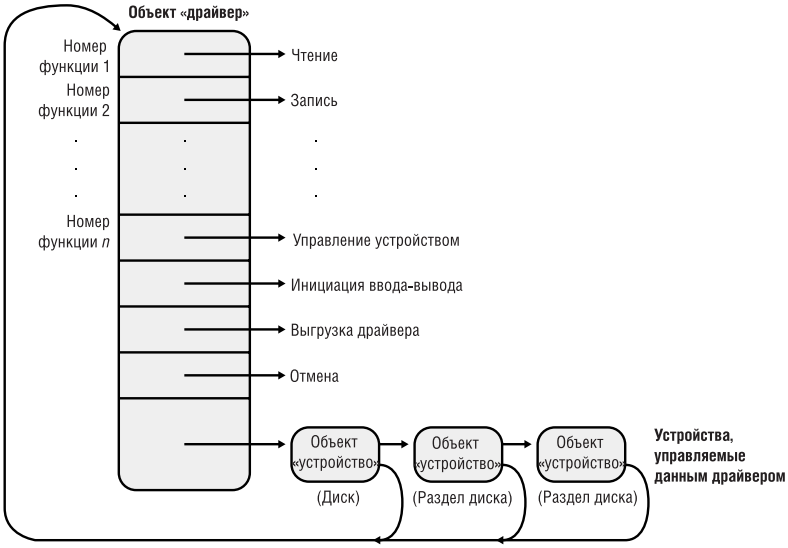


Рис. 9-6. *Объект «драйвер»*

ЭКСПЕРИМЕНТ: исследуем объекты «драйвер» и «устройство»

Эти объекты можно исследовать с помощью команд *!drvobj* и *!devobj* отладчика ядра. Следующий пример относится к объекту «драйвер» для драйверов класса «клавиатура». У этого объекта имеется единственный объект «устройство».

```
kd> !drvobj kbdclass
Driver object (81869cb0) is for:
 \Driver\Kbdclass
```

```
Driver Extension List: (id , addr)

Device Object list:
81869310

kd> !devobj 81869310
Device object (81869310) is for:
 KeyboardClass0 \Driver\Kbdclass DriverObject 81869cb0
Current Irp a57a0e90 RefCount 0 Type 0000000b Flags 00002044
DevExt 818693c8 DevObjExt 818694b8
ExtensionFlags (0000000000)
AttachedDevice (Upper) 818691e0 \Driver\Ctrl2cap
AttachedTo (Lower) 81869500 \Driver\i8042prt
Device queue is busy - Queue empty.
```

Заметьте, что команда *!devobj* заодно сообщает адреса и имена любых объектов «устройство», поверх которых размещен просматриваемый вами объект (строка *AttachedTo*), а также объектов «устройство», размещенных над указанным объектом (строка *AttachedDevice*).

Использование объектов для регистрации информации о драйверах означает, что диспетчеру ввода-вывода не нужно знать никаких деталей реализации драйверов. Он просто находит драйвер по указателю, тем самым позволяя легко загружать новые драйверы и обеспечивая их переносимость. Кроме того, представление устройств и драйверов разными объектами упрощает подсистеме ввода-вывода закрепление драйверов за дополнительными устройствами, которые появляются при изменении конфигурации системы.

Открытие устройств

Объекты «файл» являются структурами режима ядра, которые точно соответствуют определению объектов в Windows: это системные ресурсы, доступные для совместного использования двум или нескольким процессам, у них могут быть имена, их безопасность обеспечивается моделью защиты объектов, и они поддерживают синхронизацию. Хотя большинство разделяемых ресурсов в Windows базируется в памяти, основная часть ресурсов, с которыми имеет дело подсистема ввода-вывода, размещается на физических устройствах или представляет их. Несмотря на эту разницу, операции над совместно используемыми ресурсами подсистемы ввода-вывода осуществляются как над объектами.

Объекты «файл» — представление ресурсов в памяти, которое обеспечивает чтение и запись данных в эти ресурсы. В таблице 9-1 перечислены некоторые атрибуты объектов «файл». Описание и размеры полей см. в определении структуры FILE_OBJECT в Ntddk.h.

Таблица 9-1. *Атрибуты объектов «файл»*

Атрибут	Описание
Имя файла	Идентифицирует физический файл, на который ссылается объект «файл»
Текущее смещение в байтах	Идентифицирует текущий адрес в файле (действителен только для синхронного ввода-вывода)
Режимы разделения	Указывает, могут ли другие потоки открывать файл для чтения, записи или удаления, пока им пользуется текущий поток
Флаги режима открытия	Указывают тип ввода-вывода — синхронный или асинхронный, кэшируемый или некэшируемый, с последовательным или прямым доступом
Указатель на объект «устройство»	Указывает тип устройства, на котором находится файл
Указатель на блок параметров тома (VPB)	Указывает том, или раздел, на котором находится файл
Указатель на указатели объекта «раздел»	Указывает корневую структуру, описывающую проецируемый файл
Указатель на закрытую карту кэша	Идентифицирует части файла, кэшируемые диспетчером кэша, и их местонахождение в кэше

ЭКСПЕРИМЕНТ: просмотр структуры данных объекта «файл»

Вы можете просмотреть содержимое этой структуры с помощью команды *dt* отладчика ядра:

```
kd> dt nt!_file_object
nt!_FILE_OBJECT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 DeviceObject     : Ptr32 _DEVICE_OBJECT
   +0x008 Vpb              : Ptr32 _VPB
   +0x00c FsContext        : Ptr32 Void
   +0x010 FsContext2       : Ptr32 Void
   +0x014 SectionObjectPointer : Ptr32 _SECTION_OBJECT_POINTERS
   +0x018 PrivateCacheMap  : Ptr32 Void
   +0x01c FinalStatus      : Int4B
   +0x020 RelatedFileObject : Ptr32 _FILE_OBJECT
   +0x024 LockOperation    : UChar
   +0x025 DeletePending    : UChar
   +0x026 ReadAccess       : UChar
   +0x027 WriteAccess      : UChar
   +0x028 DeleteAccess     : UChar
   +0x029 SharedRead       : UChar
   +0x02a SharedWrite      : UChar
   +0x02b SharedDelete     : UChar
   +0x02c Flags            : Uint4B
   +0x030 FileName         : _UNICODE_STRING
```

```
+0x038 CurrentByteOffset : _LARGE_INTEGER
+0x040 Waiters           : Uint4B
+0x044 Busy              : Uint4B
+0x048 LastLock          : Ptr32 Void
+0x04c Lock              : _KEVENT
+0x05c Event             : _KEVENT
+0x06c CompletionContext : Ptr32 _IO_COMPLETION_CONTEXT
```

Когда поток открывает файл или простое устройство, диспетчер ввода-вывода возвращает описатель объекта «файл». Рис. 9-7 иллюстрирует, что происходит при открытии файла.

В этом примере С-программа (1) вызывает из стандартной библиотеки функцию *fopen*, которая в свою очередь вызывает Windows-функцию *CreateFile* (2). Далее DLL подсистемы Windows (в данном случае — Kernel32.dll) вызывает функцию *NtCreateFile* из Ntdll.dll (3). Эта функция в Ntdll.dll содержит соответствующие команды, вызывающие переход в режим ядра в диспетчер системных сервисов, который и обращается к настоящей функции *NtCreateFile* (4) из Ntoskrnl.exe (о диспетчеризации системных сервисов см. главу 3).

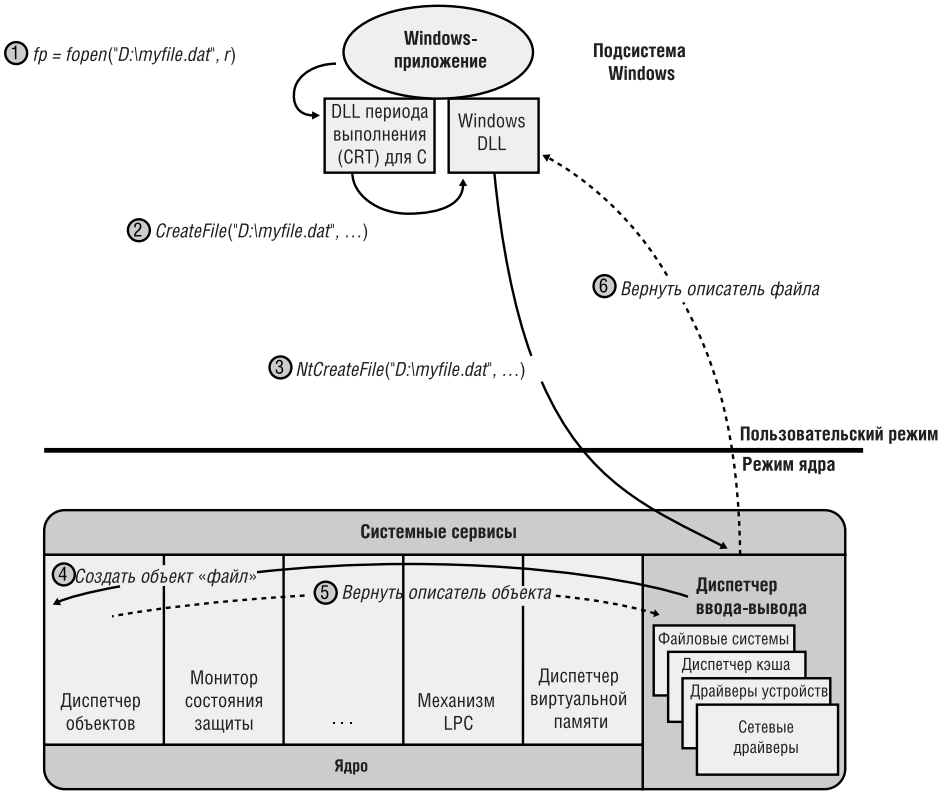


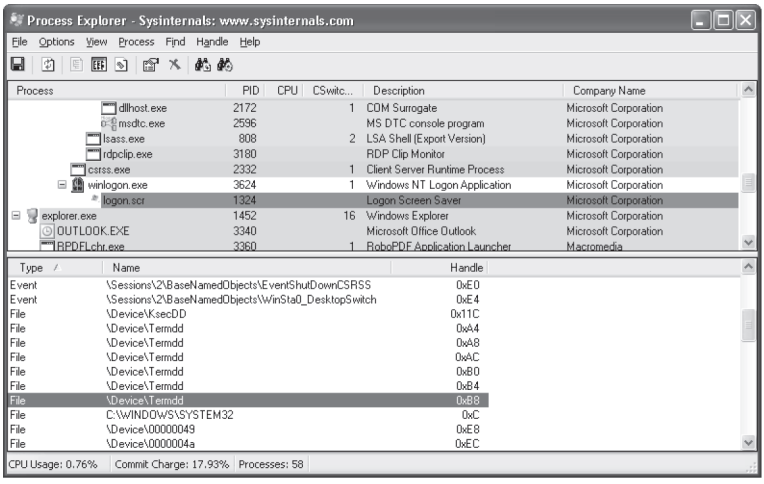
Рис. 9-7. *Открытие объекта «файл»*

ПРИМЕЧАНИЕ Объекты «файл» представляют открытые экземпляры файлов, а не сами файлы. В отличие от UNIX-систем, где используются vnode, в Windows представление файла не определено — системные драйверы Windows определяют свои представления.

Как и другие объекты исполнительной системы, файлы защищаются дескрипторами защиты, которые содержат список управления доступом (ACL). Чтобы выяснить, позволяет ли ACL файла получить процессу доступ того типа, который был запрошен его потоком, диспетчер ввода-вывода обращается к системе защиты. Если да, диспетчер объектов (5, 6) разрешает доступ и сопоставляет предоставленные права доступа с описателем файла, возвращаемым потоку. Если этому или другому потоку того же процесса понадобятся дополнительные операции с файлом, не указанные в исходном запросе, ему придется открыть новый описатель, по которому тоже будет проведена проверка прав доступа (подробнее о защите объектов см. главу 8).

ЭКСПЕРИМЕНТ: просмотр описателей устройств

У любого процесса, открывшего описатель какого-либо устройства, в таблице описателей появляется объект «файл», соответствующий открытому экземпляру. Для просмотра таких описателей вполне годится Process Explorer: выберите процесс и пометьте Show Lower Pane в меню View и Handles в подменю Lower Pane View меню View. Отсортируйте по столбцу Type и прокрутите содержимое до того места, где показываются описатели, представляющие объекты «файл»; они помечаются как «File».



В данном примере у процесса Csrss имеются открытые описатели объектов «файл», которые представляют открытые экземпляры устройств и получают автоматически генерируемые имена, а также описатели объектов «файл», принадлежащих драйверу службы терминалов. Чтобы просмотреть конкретный объект «файл» в отладчике ядра,

сначала определите адрес этого объекта. Следующая команда выводит сведения о выделенном на иллюстрации описателе (0xB8), который принадлежит процессу Csrss.exe с идентификатором 2332 (0x91c):

```
0: kd> !handle b8 f 91c
processor number 0
Searching for Process with Cid == 91c
PROCESS 86a6c020  SessionId: 0  Cid: 091c
    Peb: 7ffde000  ParentCid: 028c
    DirBase: 1158a000  ObjectTable: e1b5d080  HandleCount: 643.
    Image: csrss.exe

New version of handle table at e2b44000 with 643 Entries in use
00B8: Object: 866ae9e8  GrantedAccess: 0012019f
Object: 866ae9e8  Type: (86fe8ad0) File
    ObjectHeader: 866ae9d0
        HandleCount: 1  PointerCount: 3
```

Поскольку это объект «файл», вы можете получить информацию о нем командой *!fileobj*:

```
0: kd> !fileobj 866ae9e8

File object name:

Device Object: 0x86d0c8b8   \Driver\TermDD
Vpb is NULL

Flags:  0x40000
        Handle Created

FsContext: 0x866a31d8   FsContext2: 0x00000001
CurrentByteOffset: 0
```

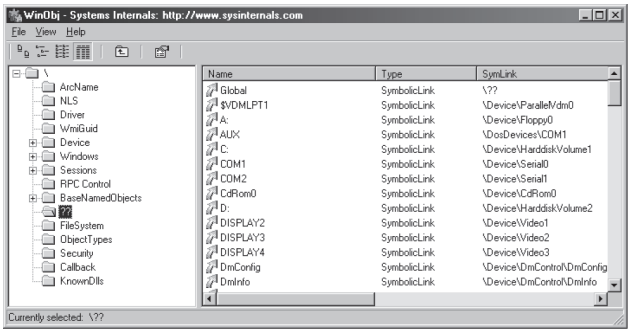
Поскольку объект «файл» — представление разделяемого ресурса в памяти, а не сам ресурс, он отличается от остальных объектов исполнительной системы. Объект «файл» содержит лишь данные, уникальные для описателя объекта, тогда как собственно файл — совместно используемые данные или текст. Всякий раз, когда поток открывает описатель файла, создается новый объект «файл» с новым набором атрибутов, специфичным для этого описателя. Например, атрибут «текущее смещение в байтах» определяет место в файле, где будут записаны или считаны следующие данные по текущему описателю. У каждого описателя одного и того же файла свое значение атрибута «текущее смещение в байтах». Кроме того, объект «файл» уникален для процесса; исключение составляют случаи, когда процесс дублирует описатель файла для передачи другому процессу (через Windows-функцию *DuplicateHandle*) или когда дочерний процесс наследует описатель файла от родительского. Только в этих двух случаях процессы располагают отдельными описателями, которые ссылаются на один и тот же объект «файл».

Описатель файла уникален для процесса, но определяемый им физический ресурс — нет. Поэтому, как и при доступе к любому другому общему ресурсу, потоки должны синхронизировать свое обращение к совместно используемым файлам, каталогам и устройствам. Если, например, поток что-то записывает в файл, то при открытии описателя файла он должен указать монопольный доступ для записи, чтобы другие потоки не могли записывать в этот файл одновременно с первым потоком. В качестве альтернативы поток может заблокировать на время записи часть файла с помощью Windows-функции *LockFile*.

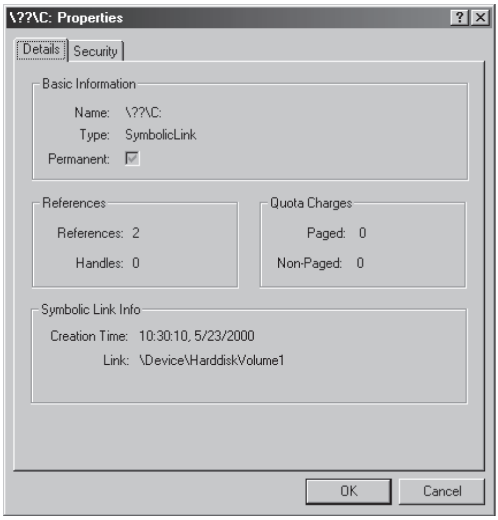
Имя открываемого файла включает имя объекта «устройство», который представляет устройство, содержащее файл. Например, \Device\Floppy0\Myfile.dat ссылается на файл Myfile.dat на дискете в дисководе A. Подстрока «\Device\Floppy0» является внутренним именем объекта «устройство», представляющего данный дисковод. При открытии файла Myfile.dat диспетчер ввода-вывода создает объект «файл», сохраняет в нем указатель на объект «устройство» Floppy0 и возвращает описатель файла вызывающему потоку. Впоследствии, когда вызывающий поток воспользуется описателем файла, диспетчер ввода-вывода сможет обращаться непосредственно к объекту Floppy0. Учтите, что внутренние имена устройств неприменимы в Windows-приложениях. Для них имена устройств должны находиться в специальном каталоге пространства имен диспетчера объектов — \?? (в Windows 2000) или \Global?? (в Windows XP и Windows Server 2003). В этом каталоге содержатся символьные ссылки на внутренние имена устройств. За создание ссылок в этом каталоге отвечают драйверы устройств. Вы можете просмотреть и даже изменить эти ссылки программным способом, через Windows-функции *QueryDosDevice* и *DefineDosDevice*.

ЭКСПЕРИМЕНТ: просмотр сопоставлений Windows-имен устройств внутренним именам устройств

Утилита Winobj (*www.sysinternals.com*) позволяет исследовать символьные ссылки, определяющие пространство Windows-имен устройств. Запустите Winobj и выберите каталог \?? в Windows 2000 или \Global?? в Windows XP либо Windows Server 2003.



Обратите внимание на символьные ссылки справа. Попробуйте дважды щелкнуть устройство C: — вы должны увидеть нечто вроде того, что показано ниже.



C: представляет собой символьную ссылку на внутреннее устройство с именем \Device\HarddiskVolume1, или на первый том первого жесткого диска в системе. Элемент COM1, показанный Winobj, — символьная ссылка на \Device\Serial0 и т. д. Попробуйте создать собственные ссылки командой *subst* в командной строке.

Обработка ввода-вывода

Теперь, когда мы рассмотрели структуру и типы драйверов, а также поддерживающие их структуры данных, давайте обсудим то, как запросы ввода-вывода проходят по системе. Обработка запросов ввода-вывода включает несколько предсказуемых этапов. Наборы операций, выполняемых на этих этапах, варьируются в зависимости от того, какой драйвер управляет устройством, которому адресован запрос, — одно- или многоуровневый. Но обработка зависит и от того, какой ввод-вывод запрошен — синхронный или асинхронный. Так что для начала мы рассмотрим эти два типа ввода-вывода и уже после этого перейдем к другим темам.

Типы ввода-вывода

Приложения могут выдавать запросы ввода-вывода различных типов, например синхронного, асинхронного, проецируемого (данные с устройства проецируются на адресное пространство приложения для доступа через виртуальную память приложения, а не API-функции ввода-вывода), а также ввода-вывода, при котором данные передаются между устройством и непрерывны-

ми буферами приложения за один запрос. Более того, диспетчер ввода-вывода позволяет драйверам реализовать специфический интерфейс ввода-вывода, что нередко обеспечивает сокращение числа IRP, необходимых для обработки ввода-вывода. В этом разделе мы рассмотрим все типы ввода-вывода.

Синхронный и асинхронный ввод-вывод

Большинство операций ввода-вывода приложений являются *синхронными*, т. е. приложение ждет, когда устройство выполнит передачу данных и вернет код статуса по завершении операции ввода-вывода. После этого программа продолжает работу и немедленно использует полученные данные. В таком простейшем варианте Windows-функции *ReadFile* и *WriteFile* выполняются синхронно. Перед возвратом управления они должны завершить операцию ввода-вывода.

Асинхронный ввод-вывод позволяет приложению выдать запрос на ввод-вывод и продолжить выполнение, не дожидаясь передачи данных устройством. Этот тип ввода-вывода увеличивает эффективность работы приложения, позволяя заниматься другими задачами, пока выполняется операция ввода-вывода. Для использования асинхронного ввода-вывода вы должны указать при вызове *CreateFile* флаг FILE_FLAG_OVERLAPPED. Конечно, инициировав операцию асинхронного ввода-вывода, поток должен соблюдать осторожность и не обращаться к запрошенным данным до их получения от устройства. Следовательно, поток должен синхронизировать свое выполнение с завершением обработки запроса на ввод-вывод, отслеживая описатель синхронизирующего объекта (которым может быть событие, порт завершения ввода-вывода или сам объект «файл»), который по окончании ввода-вывода перейдет в свободное состояние.

Независимо от типа запроса операции ввода-вывода, инициированные драйвером в интересах приложения, выполняются асинхронно, т. е. после выдачи запроса драйвер устройства возвращает управление подсистеме ввода-вывода. А когда она вернет управление приложению, зависит от типа запроса. Схема управления при инициации операции чтения показана на рис. 9-8. Заметьте, что ожидание зависит от состояния флага перекрытия в объекте «файл» и реализуется функцией *NtReadFile* в режиме ядра.

Вы можете проверить статус незавершенной операции асинхронного ввода-вывода вызовом Windows-функции *HasOverlappedIoCompleted*. При использовании портов завершения ввода-вывода с той же целью можно вызывать *GetQueuedCompletionStatus*.

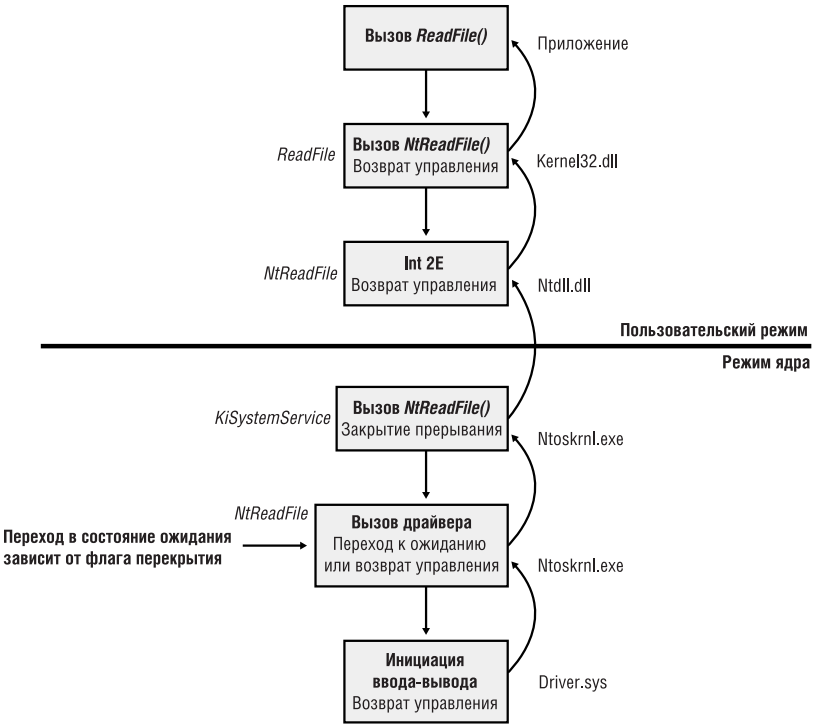


Рис. 9-8. *Схема управления при операции ввода-вывода*

Быстрый ввод-вывод

Быстрый ввод-вывод (fast I/O) — специальный механизм, который позволяет подсистеме ввода-вывода напрямую, не генерируя IRP, обращаться к драйверу файловой системы или диспетчеру кэша (быстрый ввод-вывод описывается в главах 11 и 12). Драйвер регистрирует свои точки входа для быстрого ввода-вывода, записывая их адреса в структуру, на которую ссылается указатель PFAST_IO_DISPATCH его объекта «драйвер».

ЭКСПЕРИМЕНТ: просмотр процедур быстрого ввода-вывода, зарегистрированных драйвером

Список процедур быстрого ввода-вывода, зарегистрированных драйвером в своем объекте «драйвер», выводит команда *!drvobj* отладчика ядра. Но такие процедуры обычно имеют смысл только для драйверов файловой системы. Ниже показан список процедур быстрого ввода-вывода для объекта драйвера файловой системы NTFS.

```
kd> !drvobj \filesystem\ntfs 2
Driver object (8a4372a0) is for:
 \FileSystem\Ntfs
DriverEntry:   f7bd7398 Ntfs!DriverEntry
```

см. след. стр.

```
DriverStartIo: 00000000
DriverUnload:  00000000

Dispatch routines:
[00] IRP_MJ_CREATE                      f7b76390        Ntfs!NtfsFsdCreate

Fast I/O routines:
FastIoCheckIfPossible                   f7b74a0b
Ntfs!NtfsFastIoCheckIfPossible
FastIoRead                              f7b77bbc        Ntfs!NtfsCopyReadA
FastIoWrite                             f7b8a9cc        Ntfs!NtfsCopyWriteA
FastIoQueryBasicInfo                    f7b7cd5e
Ntfs!NtfsFastQueryBasicInfo
FastIoQueryStandardInfo                 f7b7779e
Ntfs!NtfsFastQueryStdInfo
FastIoLock                              f7b8b738        Ntfs!NtfsFastLock
FastIoUnlockSingle                      f7b8b66c
Ntfs!NtfsFastUnlockSingle
FastIoUnlockAll                         f7ba5cd6
Ntfs!NtfsFastUnlockAll
FastIoUnlockAllByKey                    f7bcdab2
Ntfs!NtfsFastUnlockAllByKey
AcquireFileForNtCreateSection           f7b77771
Ntfs!NtfsAcquireForCreateSection
ReleaseFileForNtCreateSection           f7b77758
Ntfs!NtfsReleaseForCreateSection
FastIoQueryNetworkOpenInfo              f7bbec06
Ntfs!NtfsFastQueryNetworkOpenInfo
AcquireForModWrite                      f7b8663d
Ntfs!NtfsAcquireFileForModWrite
MdlRead                                 f7bbed20        Ntfs!NtfsMdlReadA
```

Как показывает вывод, NTFS зарегистрировала свою процедуру *NtfsFastIoCheckIfPossible* как элемент *FastIoCheckIfPossible* списка процедур быстрого ввода-вывода. По имени этого элемента можно догадаться, что диспетчер ввода-вывода вызывает эту функцию перед выдачей запроса на быстрый ввод-вывод и в ответ драйвер сообщает, возможны ли операции быстрого ввода-вывода применительно к данному файлу.

Ввод-вывод в проецируемые файлы и кэширование файлов

Ввод-вывод в проецируемые файлы (mapped file I/O) — важная функция подсистемы ввода-вывода, поддерживаемая ею совместно с диспетчером памяти (о проецируемых файлах см. главу 7). Термин «ввод-вывод в проецируемые файлы» относится к возможности интерпретировать файл на диске как часть виртуальной памяти процесса. Программа может обращаться к такому файлу как к большому массиву, не прибегая к буферизации или дисковому вводу-выводу. При доступе программы к памяти диспетчер памяти ис-

пользует свой механизм подкачки для загрузки нужной страницы из дискового файла. Если программа изменяет какие-то данные в своем виртуальном адресном пространстве, диспетчер памяти записывает эти данные обратно в дисковый файл в ходе обычной операции подкачки страниц.

Ввод-вывод в проецируемые файлы доступен в пользовательском режиме через Windows-функции *CreateFileMapping* и *MapViewOfFile*. В самой операционной системе такой ввод-вывод используется при выполнении важных операций — при кэшировании файлов и активизации образов (загрузке и запуске исполняемых программ). Другим потребителем этого типа ввода-вывода является диспетчер кэша. Файловые системы обращаются к диспетчеру кэша для проецирования файлов данных на виртуальную память, что ускоряет работу программ, интенсивно использующих ввод-вывод. По мере использования файла диспетчер памяти подгружает в память страницы, к которым производится обращение. Если большинство систем кэширования выделяет для кэширования фиксированную область памяти, то кэш Windows расширяется или сокращается в зависимости от объема свободной памяти. Такое изменение размера кэша возможно благодаря тому, что диспетчер кэша опирается на соответствующую поддержку со стороны диспетчера памяти (см. главу 7). Используя преимущества подсистемы подкачки страниц диспетчера памяти, диспетчер кэша избегает дублирования работы, уже проделанной диспетчером памяти. (Внутреннее устройство диспетчера кэша рассматривается в главе 11.)

Ввод-вывод по механизму «scatter/gather»

Windows также поддерживает особый вид высокопроизводительного ввода-вывода с использованием механизма «scatter/gather»; он доступен через Windows-функции *ReadFileScatter* и *WriteFileGather*. Эти функции позволяют приложению в рамках одной операции считывать или записывать данные из нескольких буферов в виртуальной памяти в непрерывную область дискового файла, а не выдавать отдельный запрос ввода-вывода для каждого буфера. Чтобы задействовать такой ввод-вывод, вы должны открыть файл для некэшируемого асинхронного (перекрывающегося) ввода-вывода и выровнять пользовательские буферы по границам страниц. Более того, если ввод-вывод направлен на устройство массовой памяти, то передаваемые данные нужно выровнять по границам секторов устройства, а их объем должен быть кратен размеру сектора.

Пакеты запроса ввода-вывода

Пакет запроса ввода-вывода (I/O request packet, IRP) хранит информацию, нужную для обработки запроса на ввод-вывод. Когда поток вызывает сервис ввода-вывода, диспетчер ввода-вывода создает IRP для представления операции в процессе ее выполнения подсистемой ввода-вывода. По возможности диспетчер ввода-вывода выделяет память под IRP в одном из двух ассоциативных списков IRP, индивидуальных для каждого процессора и хранящихся в пуле неподкачиваемой памяти. Ассоциативный список малых IRP (small-

IRP look-aside list) хранит IRP с одним блоком стека (об этих блоках — чуть позже), а ассоциативный список больших IRP (large-IRP look-aside list) — IRP с несколькими блоками стека. По умолчанию в IRP второго списка содержится 8 блоков стека, но раз в минуту система варьирует это число на основании того, сколько блоков было запрошено. Если для IRP требуется больше блоков стека, чем имеется в ассоциативном списке больших IRP, диспетчер ввода-вывода выделяет память под IRP из пула неподкачиваемой памяти. После создания и инициализации IRP диспетчер ввода-вывода сохраняет в IRP указатель на объект «файл» вызывающего потока.

ПРИМЕЧАНИЕ DWORD-параметр реестра HKLM\System\CurrentControlSet\Session Manager\I/O System\LargIrpStackLocations (если он определен) указывает, сколько блоков стека содержится в IRP, которые хранятся в ассоциативном списке больших IRP.

На рис. 9-9 показан пример запроса ввода-вывода, демонстрирующий взаимосвязи между IRP и объектами «файл», «устройство» и «драйвер». Данный пример относится к запросу ввода-вывода, который адресован одноуровневому драйверу, но большинство операций ввода-вывода гораздо сложнее, так как в их выполнении участвует не один, а несколько многоуровневых драйверов. (Этот случай мы рассмотрим позже.)

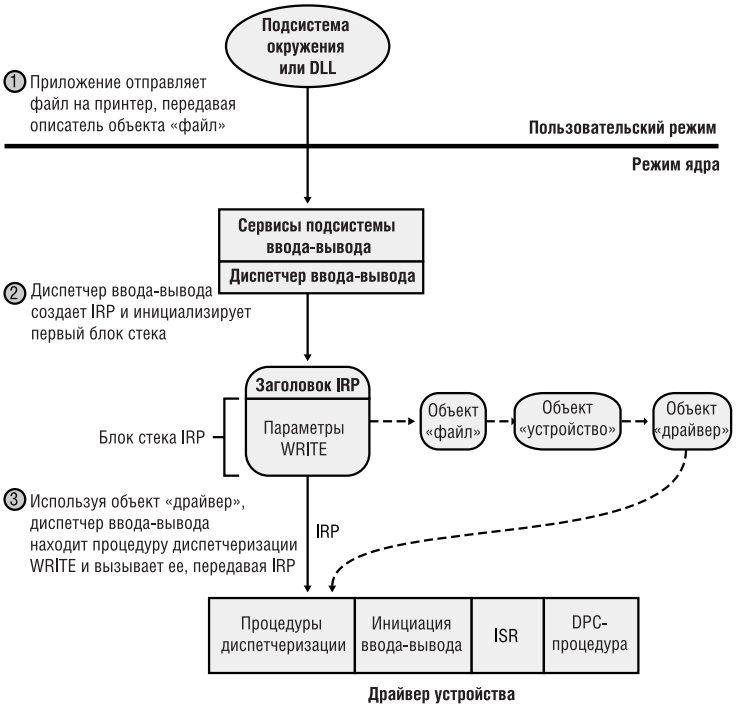


Рис. 9-9. *Структуры данных, участвующие в обработке запроса на ввод-вывод, адресованного одноуровневому драйверу*

Блок стека IRP

IRP состоит из двух частей: фиксированного заголовка (часто называемого телом IRP) и одного или нескольких блоков стека. Фиксированная часть содержит такую информацию, как тип и размер запроса, указатель на буфер в случае буферизованного ввода-вывода, данные о состоянии, изменяющиеся по мере обработки запроса, а также сведения о том, является запрос синхронным или асинхронным. Блок стека IRP (IRP stack location) содержит номер функции (состоящий из основного и дополнительного номеров), параметры, специфичные для функции, и указатель на объект «файл» вызывающего потока. *Основной номер функции* (major function code) идентифицирует принадлежащую драйверу процедуру диспетчеризации, которую диспетчер ввода-вывода вызывает при передаче IRP драйверу. Необязательный *дополнительный номер функции* (minor function code) иногда используется как модификатор основного номера. В командах управления электропитанием и Plug and Play всегда указывается дополнительный номер функции.

В большинстве драйверов процедуры диспетчеризации определены только для подмножества основных функций, т. е. функций, предназначенных для создания/открытия, записи, чтения, управления вводом-выводом на устройстве, управления электропитанием, операций Plug and Play, System (для WMI-команд) и закрытия. Драйверы файловой системы определяют функции для всех (или почти всех) точек входа. Диспетчер ввода-вывода записывает в точки входа, не заполненные драйверами, указатели на свою функцию *IopInvalidDeviceRequest*. Эта функция возвращает вызывающему потоку код ошибки, который уведомляет о попытке обращения к функции, не поддерживаемой данным устройством.

ЭКСПЕРИМЕНТ: исследуем процедуры диспетчеризации, принадлежащие драйверу

Вы можете получить список всех функций, определенных драйвером для своих процедур диспетчеризации. Для этого введите команду *!drvobj* отладчика ядра и после имени (или адреса) объекта «драйвер» укажите значение 7. Следующий вывод показывает, что драйверы поддерживают 28 типов IRP.

```
kd> !drvobj kbdclass 7
Driver object (8a238900) is for:
 \Driver\Kbdclass
Driver Extension List: (id , addr)

Device Object list:
8a189030  8a2501f8

DriverEntry:   f7822d22 kbdclass!DriverEntry
DriverStartIo: 00000000
DriverUnload:  00000000
```

см. след. стр.

```
Dispatch routines:
[00] IRP_MJ_CREATE                      f781fd3b
kbdclass!KeyboardClassCreate
[01] IRP_MJ_CREATE_NAMED_PIPE           804eef8e
nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                       f781ff4c
kbdclass!KeyboardClassClose
[03] IRP_MJ_READ                        f7820ba5
kbdclass!KeyboardClassRead
[04] IRP_MJ_WRITE                       804eef8e
nt!IopInvalidDeviceRequest
[05] IRP_MJ_QUERY_INFORMATION           804eef8e
nt!IopInvalidDeviceRequest
[06] IRP_MJ_SET_INFORMATION             804eef8e
nt!IopInvalidDeviceRequest
[07] IRP_MJ_QUERY_EA                    804eef8e
nt!IopInvalidDeviceRequest
[08] IRP_MJ_SET_EA                      804eef8e
nt!IopInvalidDeviceRequest
[09] IRP_MJ_FLUSH_BUFFERS               f781fcbe
kbdclass!KeyboardClassFlush
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION    804eef8e
nt!IopInvalidDeviceRequest
[0b] IRP_MJ_SET_VOLUME_INFORMATION      804eef8e
nt!IopInvalidDeviceRequest
[0c] IRP_MJ_DIRECTORY_CONTROL           804eef8e
nt!IopInvalidDeviceRequest
[0d] IRP_MJ_FILE_SYSTEM_CONTROL         804eef8e
nt!IopInvalidDeviceRequest
[0e] IRP_MJ_DEVICE_CONTROL              f7821829
kbdclass!KeyboardClassDeviceControl
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     f7821200
kbdclass!KeyboardClassPassThrough
[10] IRP_MJ_SHUTDOWN                    804eef8e
nt!IopInvalidDeviceRequest
[11] IRP_MJ_LOCK_CONTROL                804eef8e
nt!IopInvalidDeviceRequest
[12] IRP_MJ_CLEANUP                     f781fc84
kbdclass!KeyboardClassCleanup
[13] IRP_MJ_CREATE_MAILSLOT             804eef8e
nt!IopInvalidDeviceRequest
[14] IRP_MJ_QUERY_SECURITY              804eef8e
nt!IopInvalidDeviceRequest
[15] IRP_MJ_SET_SECURITY                804eef8e
nt!IopInvalidDeviceRequest
[16] IRP_MJ_POWER                       f7821f51
kbdclass!KeyboardClassPower
[17] IRP_MJ_SYSTEM_CONTROL              f7821649
```

```
kbdclass!KeyboardClassSystemControl
[18] IRP_MJ_DEVICE_CHANGE               804eef8e
nt!IopInvalidDeviceRequest
[19] IRP_MJ_QUERY_QUOTA                 804eef8e
nt!IopInvalidDeviceRequest
[1a] IRP_MJ_SET_QUOTA                   804eef8e
nt!IopInvalidDeviceRequest
[1b] IRP_MJ_PNP                         f78206c1
kbdclass!KeyboardPnP
```

Каждый IRP, пока он активен, хранится в списке IRP, сопоставленном с потоком, который выдал запрос на ввод-вывод. Это позволяет подсистеме ввода-вывода найти и отменить любые незавершенные IRP, если выдавший их поток завершается.

ЭКСПЕРИМЕНТ: просмотр незавершенных IRP потока

Команда *!thread* выводит любые IRP, сопоставленные с потоком. Запустите отладчик ядра в работающей системе и найдите процесс диспетчера управления сервисами (Services.exe) в выводе, сгенерированном командой *!process*:

```
lkd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
...
PROCESS 8a238da8  SessionId: 0  Cid: 02a8    Peb: 7ffdf000
    ParentCid: 027c   DirBase: 14fac000  ObjectTable: e1c3e008
    HandleCount: 365.
    Image: SERVICES.EXE
...
```

Теперь создайте дамп потоков для процесса, выполнив команду *!process* применительно к объекту «процесс». Вы должны увидеть множество потоков, у большинства из которых есть IRP; эти IRP отображаются в блоке IRP List информации о потоке (заметьте, что отладчик выводит лишь первые 17 IRP для потока, у которого имеется более 17 незавершенных запросов ввода-вывода):

```
kd> !process 8a238da8
PROCESS 8a238da8  SessionId: 0  Cid: 02a8    Peb: 7ffdf000
    ParentCid: 027c   DirBase: 14fac000  ObjectTable: e1c3e008
    HandleCount: 365.
    Image: SERVICES.EXE
    VadRoot 8a1be328 Vads 88 Clone 0 Private 346. Modified 37.
    Locked 0. DeviceMap e10087c0
...
     THREAD 8a124870  Cid 02a8.0338  Teb: 7ffd8000 Win32Thread:
00000000 WAIT: (WrQueue) UserMode Non-Alertable
            8a2dc620  Unknown
            8a124960  NotificationTimer
```

см. след. стр.

```
        IRP List:         8a2c2c00: (0006,0094) Flags: 00000900
                          Mdl: 00000000
            8a20f770: (0006,0094) Flags: 00000900  Mdl: 00000000
            8a437780: (0006,0094) Flags: 00000900  Mdl: 00000000
            89b1de68: (0006,0094) Flags: 00000900  Mdl: 00000000
            8a0e6058: (0006,0094) Flags: 00000900  Mdl: 00000000
            8a0f1550: (0006,0094) Flags: 00000900  Mdl: 00000000
            8a3b3c18: (0006,0094) Flags: 00000900  Mdl: 00000000
            8a429190: (0006,0094) Flags: 00000900  Mdl: 00000000
            8a49f008: (0006,0094) Flags: 00000900  Mdl: 00000000
            8a227bc0: (0006,0094) Flags: 00000900  Mdl: 00000000
...
```

Выберите IRP и просмотрите его командой *!irp*:

```
lkd> !irp 8a2c2c00
Irp is active with 1 stacks 1 is current (= 0x8a2c2c70)
 No Mdl Thread 8a124870:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
>[  3, 0]   0  1 8a0e5680 8a26e4b8 00000000-00000000    pending
           \Driver\Npfs
            Args: 00000400 00000000 00000000 00000000
```

У этого IRP основной номер функции — 3, что соответствует IRP_MJ_READ. Он содержит один блок стека и адресован устройству, принадлежащему драйверу Npfs (Named Pipe File System). (Информацию о Npfs см. в главе 13.)

Управление буфером IRP

Когда приложение или драйвер устройства неявно создает IRP с помощью системного сервиса *NtReadFile*, *NtWriteFile* или *NtDeviceIoControlFile* (этим сервисам соответствуют Windows-функции *ReadFile*, *WriteFile* и *DeviceIoControl*), диспетчер ввода-вывода определяет, должен ли он участвовать в управлении буферами ввода и вывода вызывающего потока. Диспетчер ввода-вывода поддерживает три вида управления буферами.

- **Буферизованный ввод-вывод (buffered I/O)** Диспетчер ввода-вывода выделяет в пуле неподкачиваемой памяти буфер, равный по размеру буферу вызывающего потока. Создавая IRP при операциях записи, диспетчер ввода-вывода копирует данные из буфера вызывающего потока в выделенный буфер. Завершая обработку IRP при операциях чтения, диспетчер ввода-вывода копирует данные из выделенного буфера в пользовательский буфер и освобождает выделенный буфер.
- **Прямой ввод-вывод (direct I/O)** Создавая IRP, диспетчер ввода-вывода блокирует пользовательский буфер в памяти (делает его неподкачиваемым). Закончив работу с IRP, диспетчер ввода-вывода разблокирует буфер. Диспетчер хранит описание этой памяти в форме MDL (memory descriptor list). MDL указывает объем физической памяти, занятой буфером (подроб-

нее о MDL см. Windows DDK). Устройствам, использующим DMA (прямой доступ к памяти), требуется лишь физическое описание буфера, поэтому таким устройствам достаточно MDL. (Устройства, поддерживающие DMA, передают данные в память компьютера напрямую, не используя процессор.) Но, если драйверу нужен доступ к содержимому буфера, он может спроецировать его на системное адресное пространство.

- **Ввод-вывод без управления (neither I/O)** Диспетчер ввода-вывода не участвует в управлении буферами. Ответственность за управление ими возлагается на драйвер устройства.

При любом типе управления буферами диспетчер ввода-вывода помещает в IRP ссылки на буферы ввода и вывода. Тип управления буферами, реализуемого диспетчером ввода-вывода, зависит от типа управления, запрошенного драйвером для операций конкретного типа. Драйвер регистрирует нужный ему тип управления буферами для операций чтения и записи в объекте «устройство», который представляет устройство. Операции управления вводом-выводом на устройстве (выполняемые *NtDeviceIoControlFile*) задаются определенными в драйвере управляющими кодами ввода-вывода. Управляющий код включает тип управления буферами со стороны диспетчера ввода-вывода при обработке IRP с данным кодом.

Когда вызывающие потоки передают запросы размером менее одной страницы (4 Кб на x86-процессорах), драйверы, как правило, используют буферизованный ввод-вывод, а для запросов большего размера — прямой. Буфер примерно равен размеру страницы, и операция копирования с применением буферизованного ввода-вывода приводит практически к тем же издержкам, что и прямой ввод-вывод, требующий блокирования памяти. Драйверы файловой системы обычно используют третий тип управления, так как при копировании данных из кэша файловой системы в буфер вызывающего потока это позволяет избавиться от издержек, связанных с управлением буферами. Но большинство драйверов не использует этот вид управления из-за того, что указатель на буфер вызывающего потока действителен лишь на то время, пока выполняется этот поток. Если драйверу нужно передать данные с устройства (или на устройство) при выполнении DPC-процедуры или ISR, он должен позаботиться о доступности данных вызывающего потока из контекста любого процесса, а значит, у буфера должен быть системный виртуальный адрес.

Драйверы, использующие ввод-вывод без управления для доступа к буферам, которые могут быть расположены в пользовательском пространстве, должны проверять, что адреса буфера действительны и не ссылаются на память режима ядра. Если они этого не делают, появляется вероятность краха системы или уязвимости в системе защиты, так как приложения получают доступ к памяти режима ядра или возможность внедрения своего кода в ядро. Функции *ProbeForRead* и *ProbeForWrite*, которые ядро предоставляет драйверам, проверяют, полностью ли умещается буфер в пользовательской части адресного пространства. Чтобы избежать краха из-за ссылки на недопустимый адрес, драйверы могут обращаться к буферам пользовательского режима из кода обработки исключений (блоков try/except), который перехваты-

вает любые попытки доступа по неправильным адресам и транслирует их в коды ошибок для передачи приложению.

Запрос ввода-вывода к одноуровневому драйверу

В этом разделе вы увидите, как обрабатывается запрос синхронного ввода-вывода к одноуровневому драйверу режима ядра. Такая обработка проходит в семь этапов.

1. Запрос на ввод-вывод передается через DLL подсистемы.
2. DLL подсистемы вызывает сервис *NtWriteFile* диспетчера ввода-вывода.
3. Диспетчер ввода-вывода создает IRP, описывающий запрос, и посылает его драйверу (в данном случае — драйверу устройства), вызывая свою функцию *IoCallDriver*.
4. Драйвер передает данные из IRP на устройство и инициирует операцию ввода-вывода.
5. Драйвер уведомляет о завершении ввода-вывода, генерируя прерывание.
6. Когда устройство завершает операцию и вызывает прерывание, драйвер устройства обслуживает прерывание.
7. Драйвер вызывает функцию *IoCompleteRequest* диспетчера ввода-вывода, чтобы уведомить его о завершении обработки IRP, и диспетчер ввода-вывода завершает данный запрос на ввод-вывод.
   Эти семь этапов показаны на рис. 9-10.

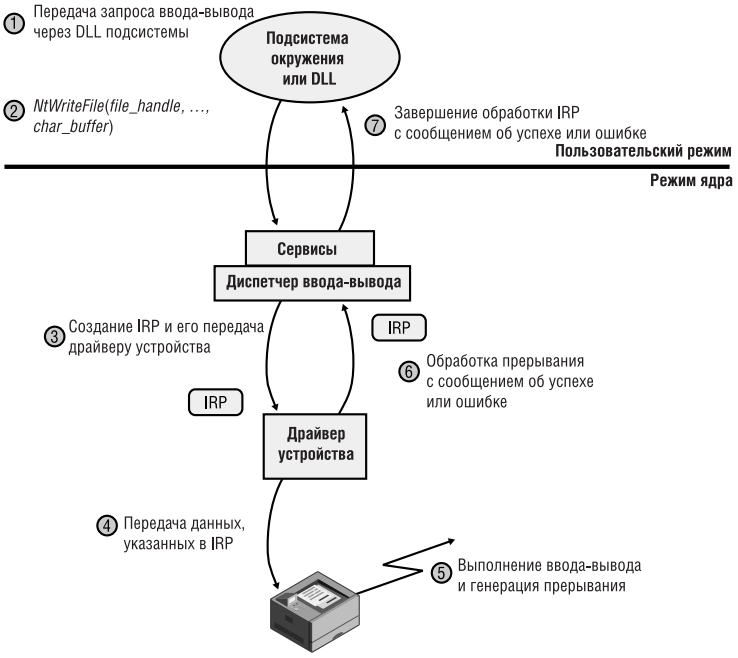


Рис. 9-10. *Обработка синхронного запроса*

Теперь, когда мы знаем, как инициируется ввод-вывод, рассмотрим обслуживание прерывания и завершение ввода-вывода.

Обслуживание прерывания

Завершая передачу данных, устройство генерирует прерывание, после чего в дело вступают ядро Windows, диспетчер ввода-вывода и драйвер устройства. На рис. 9-11 показана первая фаза этого процесса. (Механизм диспетчеризации прерываний, включая DPC, описывается в главе 3. Мы кратко повторяем этот материал, потому что DPC играют ключевую роль в обработке ввода-вывода.)

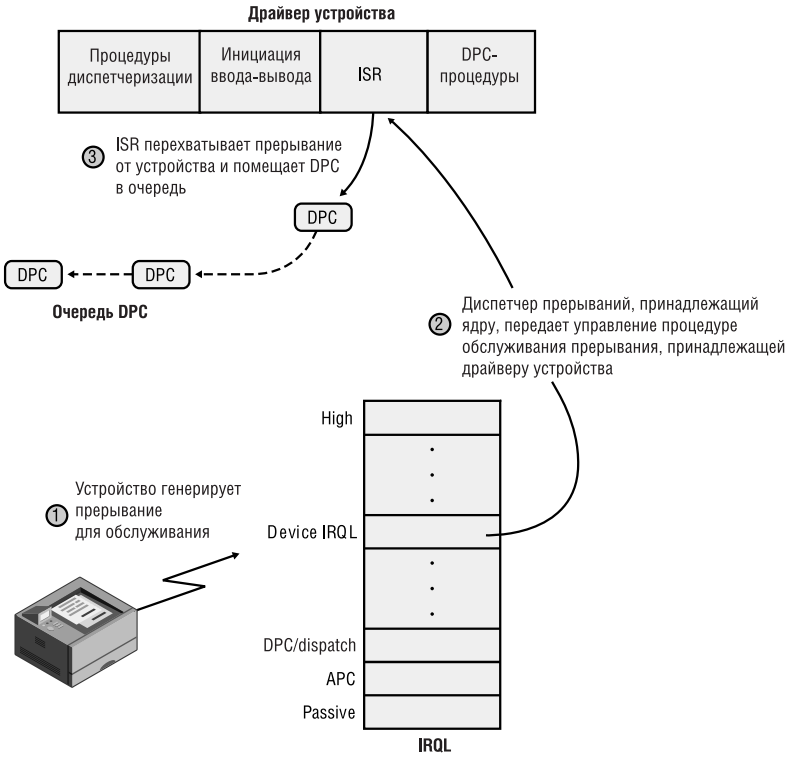


Рис. 9-11. *Обслуживание прерывания от устройства (фаза 1)*

Когда устройство генерирует прерывание, процессор передает управление обработчику ловушки ядра, который находит ISR для этого устройства по таблице диспетчеризации прерываний. ISR в Windows обычно обрабатывают прерывания от устройств в два этапа. При первом вызове ISR, как правило, остается на уровне Device IRQL ровно столько времени, сколько нужно для того, чтобы сохранить состояние устройства и запретить дальнейшие прерывания от него. После этого ISR помещает DPC в очередь и, закрыв прерывание, завершается. Впоследствии, при вызове DPC-процедуры драйвер

устройства заканчивает обработку прерывания, а затем вызывает диспетчер ввода-вывода для завершения ввода-вывода и удаления IRP. Этот драйвер также может начать выполнение следующего запроса ввода-вывода, ждущего в очереди устройства.

Преимущество выполнения большей части обработки прерываний от устройств через DPC в том, что это разрешает любые блокируемые прерывания с приоритетами от «Device IRQL» до «DPC/dispatch» — пока не началась обработка DPC, имеющего более низкий приоритет. А за счет этого удается более оперативно (чем это могло бы быть в ином случае) обслуживать прерывания среднего приоритета. Вторая фаза ввода-вывода (обработка DPC) показана на рис. 9-12.

Завершение обработки запроса на ввод-вывод

После того как DPC-процедура драйвера выполнена, до завершения запроса на ввод-вывод остается проделать кое-какую оставшуюся работу. Третья стадия обработки ввода-вывода называется *завершением ввода-вывода* (I/O completion) и начинается с вызова драйвером функции *IoCompleteRequest* для уведомления диспетчера ввода-вывода о том, что обработка запроса, указанного в IRP (и принадлежащих ему блоках стека), закончена. Действия, выполняемые на этом этапе, различны для разных операций ввода-вывода. Например, все сервисы ввода-вывода записывают результат операции в блок статуса ввода-вывода (I/O status block) — структуру данных, предоставляемую вызывающим потоком. Некоторые сервисы, выполняющие буферизованный ввод-вывод, требуют возврата данных вызывающему потоку через подсистему ввода-вывода.

В любом случае подсистема ввода-вывода должна копировать отдельные данные из системной памяти в виртуальное адресное пространство процесса, которому принадлежит вызывающий поток. Если IRP выполняется синхронно, это адресное пространство является текущим и доступно напрямую, но если IRP обрабатывается асинхронно, диспетчер ввода-вывода должен отложить завершение IRP до тех пор, пока у него не появится возможность обращаться к нужному адресному пространству. Чтобы получить доступ к виртуальному адресному пространству процесса, которому принадлежит вызывающий поток, диспетчер ввода-вывода должен передавать данные «в контексте вызывающего потока», т. е. при выполнении этого потока (иначе говоря, процесс этого потока должен быть текущим, а его адресное пространство — активным на процессоре). Эту задачу диспетчер ввода-вывода решает, ставя в очередь данному потоку APC режима ядра (рис. 9-13).

Как уже говорилось в главе 3, APC выполняется только в контексте определенного потока, а DPC — в контексте любого потока. Это означает, что DPC не затрагивает адресное пространство процесса пользовательского режима. Вспомните также, что приоритет программного прерывания у DPC выше, чем у APC.

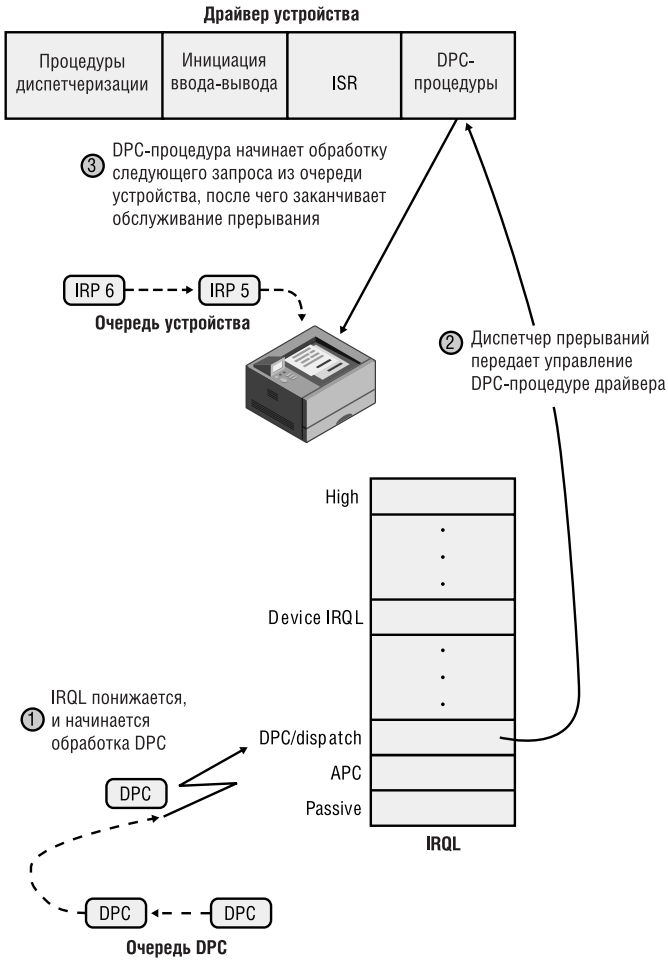


Рис. 9-12. *Обслуживание прерывания от устройства (фаза 2)*

В следующий раз, когда поток начинает выполняться при низком IRQL, ему доставляется отложенный APC. Ядро передает управление APC-процедуре диспетчера ввода-вывода, которая копирует данные (если они есть) и код возврата в адресное пространство процесса вызывающего потока, освобождает IRP, представляющий данную операцию ввода-вывода, и переводит описатель файла (и любое событие или порт завершения ввода-вывода, если таковой объект предоставлен вызывающим потоком) в свободное состояние. Теперь ввод-вывод считается завершенным. Вызывающий поток или любые другие потоки, ждущие на описателе файла (или иного объекта), выходят из состояния ожидания и переходят в состояние готовности к выполнению. Вторая фаза завершения ввода-вывода показана на рис. 9-14.

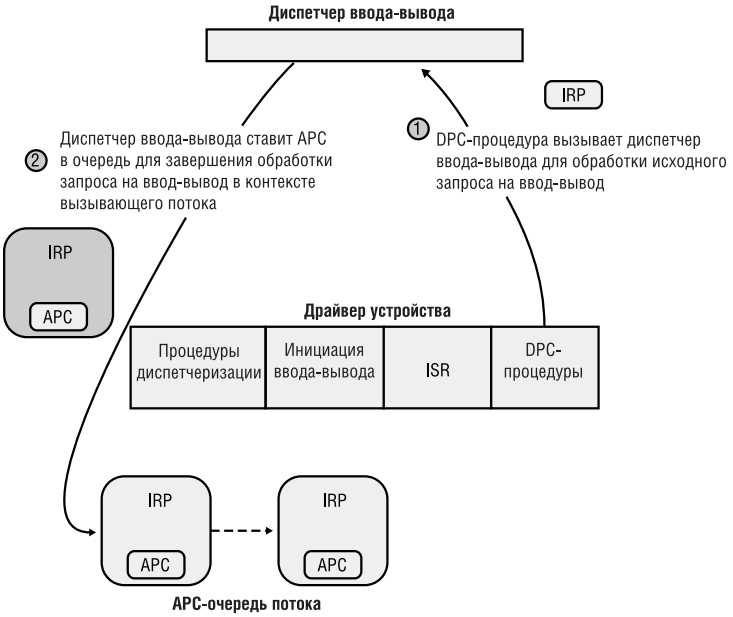


Рис. 9-13. *Завершение обработки запроса на ввод-вывод (фаза 1)*

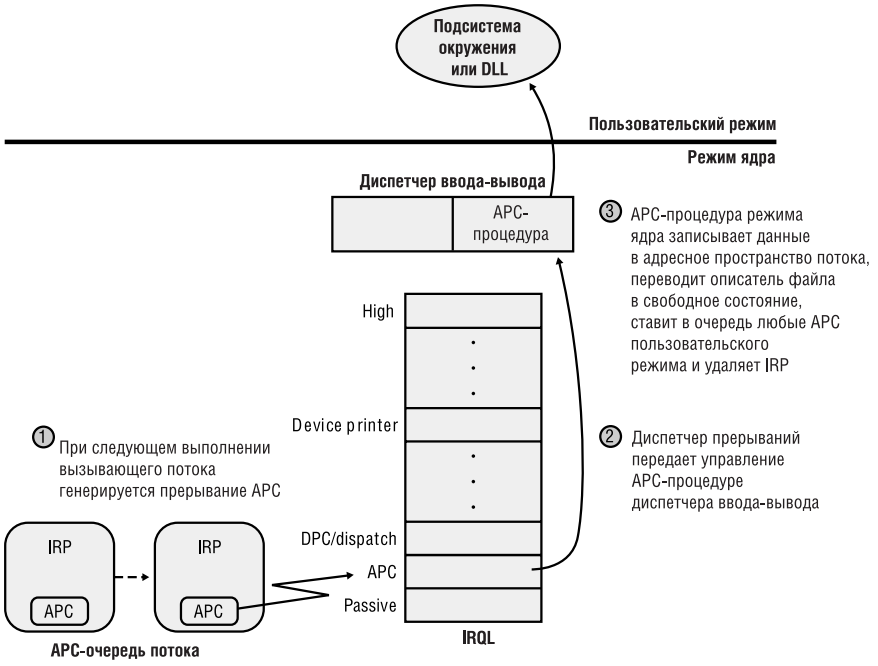


Рис. 9-14. *Завершение обработки запроса на ввод-вывод (фаза 2)*

И последнее замечание о завершении ввода-вывода. Функции асинхронного ввода-вывода *ReadFileEx* и *WriteFileEx* принимают в качестве параметра APC пользовательского режима. Если поток передаст этот параметр, то на последнем этапе диспетчер ввода-вывода направит соответствующий APC в очередь данного потока. Эта функциональность позволяет вызывающему потоку указывать процедуру, которую нужно вызывать после завершения или отмены запроса ввода-вывода. Такие APC выполняются в контексте вызывающего потока и доставляются, только если поток переходит в состояние «тревожного» ожидания (как, например, при вызове Windows-функции *SleepEx*, *WaitForSingleObjectEx* или *WaitForMultipleObjectsEx*).

Синхронизация

Драйверы должны синхронизировать свое обращение к глобальным данным и регистрам устройств в силу двух причин.

- Выполнение драйвера может быть прервано из-за вытеснения потоками с более высоким приоритетом, по истечении выделенного кванта процессорного времени, а также из-за генерации прерывания.
- В многопроцессорных системах Windows может выполнять код драйвера сразу на нескольких процессорах.

Без синхронизации данные могут быть повреждены. Например, код драйвера устройства выполняется при IRQL уровня «passive». Какая-то программа инициирует операцию ввода-вывода, в результате чего возникает аппаратное прерывание. Оно прерывает выполнение кода драйвера и активизирует его ISR. Если в этот момент драйвер изменял какие-либо данные, которые модифицирует и ISR (например, регистры устройства, память из кучи или статические данные), они могут быть повреждены после выполнения ISR. Эту проблему демонстрирует рис. 9-15.

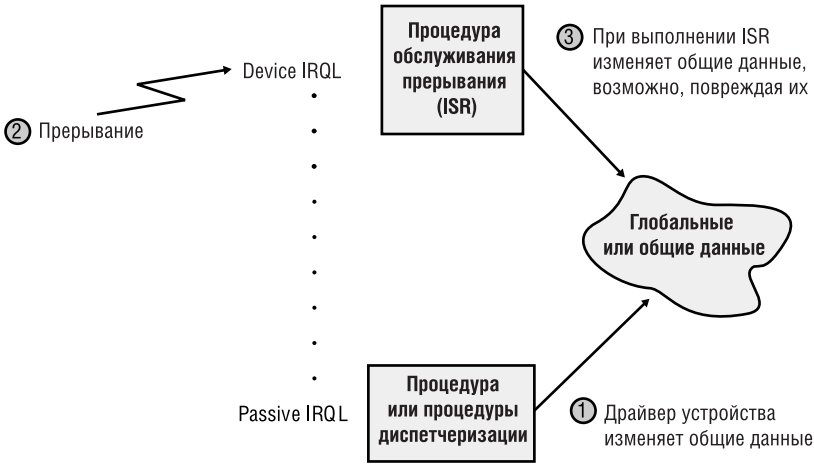


Рис. 9-15. *Пример повреждения данных, используемых драйвером устройства, в отсутствие синхронизации*

Во избежание такой ситуации драйвер, написанный для Windows, должен синхронизировать обращение к любым данным, которые он разделяет со своей ISR. Прежде чем обновлять общие данные, драйвер должен заблокировать все остальные потоки (или процессоры, если система многопроцессорная), чтобы запретить им доступ к тем же данным.

Ядро Windows предоставляет специальную синхронизирующую процедуру *KeSynchronizeExecution*, которую драйверы устройств должны вызывать при доступе к данным, разделяемым с ISR. Эта процедура не допускает выполнения ISR, пока драйвер обращается к общим данным. В однопроцессорных системах перед обновлением общих структур данных она повышает IRQL до уровня, сопоставленного с ISR. Но в многопроцессорных системах эта методика не гарантирует полной блокировки, так как код драйвера может выполняться на двух и более процессорах одновременно. Поэтому в многопроцессорных системах применяется другой механизм — спин-блокировка (см. раздел «Синхронизация ядра» главы 3). Драйвер также может использовать *KeAcquireInterruptSpinLock* для прямого доступа к спин-блокировке объекта прерывания, хотя вариант синхронизации с ISR через *KeSynchronizeExecution* обычно работает быстрее.

Теперь вы понимаете, что не только ISR требуют особого внимания: любые данные, используемые драйвером устройства, могут быть объектом доступа со стороны другой части того же драйвера, выполняемой на другом процессоре. Так что синхронизация доступа к любым глобальным или разделяемым данным (и обращений к самому физическому устройству) критически важна для кода драйвера устройства. Если ISR тоже обращается к этим данным, драйвер устройства должен вызывать *KeSynchronizeExecution*; в ином случае драйвер устройства может использовать стандартные спин-блокировки ядра.

Запрос ввода-вывода к многоуровневому драйверу

В предыдущем разделе мы рассмотрели обработку запроса на ввод-вывод, адресованного простому устройству, которое управляется единственным драйвером устройства. Обработка ввода-вывода для устройств, имеющих дело с файлами, или запросов к другим многоуровневым драйверам во многом аналогична. Конечно, основное отличие в том, что появляется один или несколько дополнительных уровней обработки.

Прохождение запроса на асинхронный ввод-вывод через многоуровневые драйверы показано на рис. 9-16. Данный пример относится к диску, управляемому файловой системой.

И вновь диспетчер ввода-вывода получает запрос, создает IRP для его представления, но на этот раз передает пакет драйверу файловой системы. С этого момента драйвер файловой системы в основном и управляет операцией ввода-вывода. В зависимости от типа запроса файловая система посылает драйверу диска тот же IRP или генерирует дополнительные IRP и передает их этому драйверу по отдельности.

ЭКСПЕРИМЕНТ: просмотр стека устройства

Команда *!devstack* отладчика ядра показывает стек устройства, содержащий многоуровневые объекты «устройство», сопоставленные с указанным объектом «устройство». В данном примере выводится стек устройства для объекта «устройство» \device\keyboardclass0, который принадлежит драйверу класса клавиатур:

```
lkd> !devstack keyboardclass0
  !DevObj   !DrvObj            !DevExt   ObjectName
  8a266d28  \Driver\Ctrl2cap   8a266de0
> 8a09a030  \Driver\Kbdclass   8a09a0e8  KeyboardClass0
  8a2672b0  \Driver\nmfilter   8a267368  00000080
  8a09ba78  \Driver\i8042prt   8a09bb30
  8a4adce0  \Driver\ACPI       8a4ab9c8  0000006b
!DevNode 8a4acee8 :
  DeviceInst is "ACPI\PNP0303\4&61f3b4b&0"
  ServiceName is "i8042prt"
```

Строка для KeyboardClass0 выделяется префиксом «>». Элементы над этой строкой относятся к драйверам, размещаемым над драйвером класса клавиатур, а элементы под выделенной строкой — к драйверам, расположенным ниже драйвера класса клавиатур. В целом, IRP передаются по стеку сверху вниз.

Файловая система скорее всего будет повторно использовать IRP, если полученный запрос можно преобразовать в единый запрос к устройству. Например, если приложение выдаст запрос на чтение первых 512 байтов из файла на дискете, файловая система FAT просто вызовет драйвер диска, попросив его считать один сектор с того места на дискете, где начинается нужный файл.

Для поддержки использования несколькими драйверами IRP содержит набор блоков стека (не путать со стеком потока). Эти блоки данных — по одному на каждый вызываемый драйвер — хранят информацию, необходимую каждому драйверу для обработки своей части запроса (например, номер функции, параметры, сведения о контексте драйвера). Как показано на рис. 9-16, по мере передачи IRP от одного драйвера другому заполняются дополнительные блоки стека. IRP можно считать аналогом стека в отношении добавления и удаления данных. Но IRP не сопоставляется ни с каким процессом, и его размер фиксирован. В самом начале операции ввода-вывода диспетчер ввода-вывода выделяет память для IRP в одном из ассоциативных списков IRP или в пуле неподкачиваемой памяти.

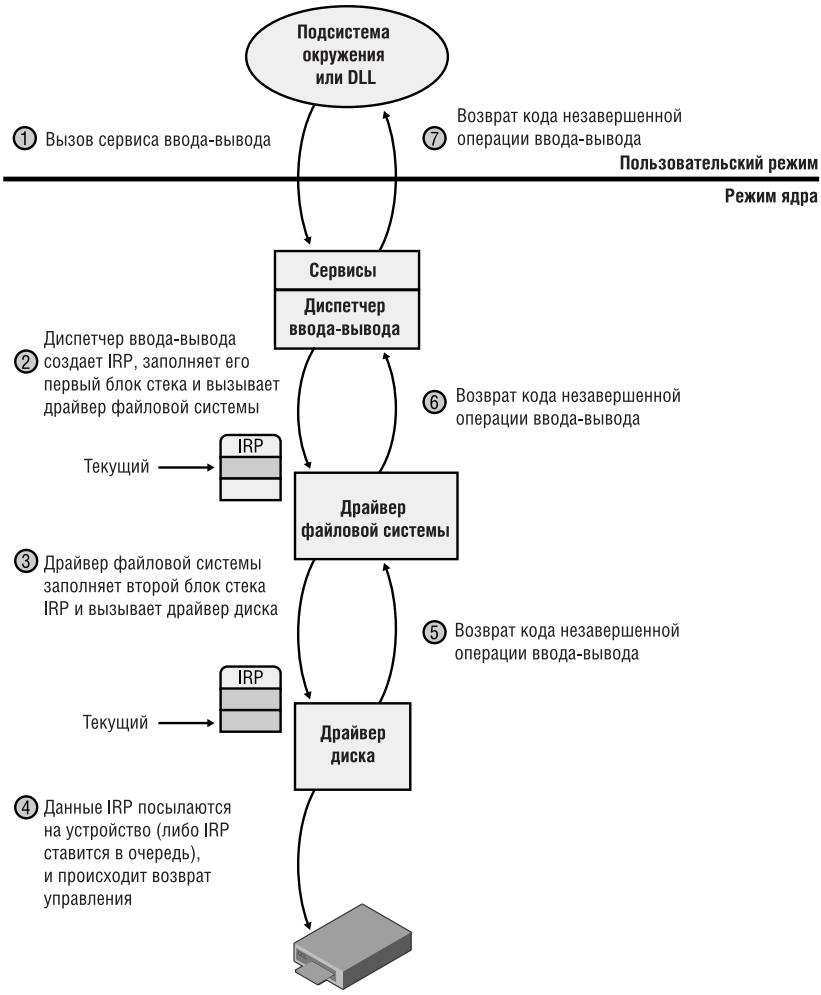


Рис. 9-16. *Обработка асинхронного запроса к многоуровневым драйверам*

ЭКСПЕРИМЕНТ: исследуем IRP

В этом эксперименте вы найдете незавершенные IRP в системе и определите тип IRP, устройство, которому он адресован, драйвер, управляющий этим устройством, поток, выдавший IRP, и процесс, к которому относится данный поток.

В любой момент в системе есть хотя бы несколько незавершенных IRP. Это вызвано тем, что существует много устройств, которым приложения могут посылать IRP, а драйвер обрабатывает запрос только при возникновении определенного события, скажем, при появлении данных. Один из примеров — чтение с сетевого устройства. Увидеть незавершенные IRP в системе позволяет команда *!irpfind* отладчика ядра:

```
kd> !irpfind
unable to get large pool allocation table - either wrong symbols
 or pool tagging is disabled

Searching NonPaged pool (82502000 : 8a502000) for Tag: Irp?

  Irp    [ Thread ] irpStack: (Mj,Mn)   DevObj  [Driver]
89695868 [00000000] Irp is complete (CurrentLocation 4 >
 StackCount 3) 0x43776f56
89712008 [8a29d7c0] irpStack: ( e, 9)  8a19e208 [ \Driver\AFD]
89716008 [8a29d7c0] irpStack: ( e, 9)  8a19e208 [ \Driver\AFD]
...
89cb3928 [8a3acbc0] irpStack: ( 3, 0)  8a09a030 [ \Driver\Kbdclass]
89cb3c88 [89cb1da8] irpStack: ( c, 2)  8a436020 [ \FileSystem\Ntfs]
89cb4640 [8a165498] irpStack: ( e, 9)  8a19e208 [ \Driver\AFD]
```

Строка, выделенная в выводе, описывает IRP, адресованный драйверу Kbdclass, так что этот IRP скорее всего был выдан потоком необработанного ввода для подсистемы Windows, принимающим ввод с клавиатуры. Изучение IRP с помощью команды *!irp* показывает следующее:

```
kd> !irp 8a1716f0
Irp is active with 3 stacks 3 is current (= 0x8a1717a8)
 No Mdl System buffer = 8a458180 Thread 8a3acbc0:
     Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                        Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                        Args: 00000000 00000000 00000000 00000000
>[  3, 0]   0  1 8a1eccb8 8a1262a8 00000000-00000000    pending
               \Driver\Kbdclass
                        Args: 00000078 00000000 00000000 00000000
```

Активный блок стека (помечаемый префиксом «>», находится в самом низу. Основной номер функции равен 3, что соответствует IRP_MJ_READ.

Следующий шаг — выяснить, какому объекту «устройство» адресован IRP. Для этого выполните команду *!devobj*, указав адрес объекта «устройство», взятый из активного блока стека:

```
kd> !devobj 8a1eccb8
Device object (8a1eccb8) is for:
 KeyboardClass1 \Driver\Kbdclass DriverObject 8a24bd78
Current Irp 00000000 RefCount 0 Type 0000000b Flags 00002044
Dacl e1ec01e4 DevExt 8a1ecd70 DevObjExt 8a1ece50
ExtensionFlags (0000000000)
```

см. след. стр.

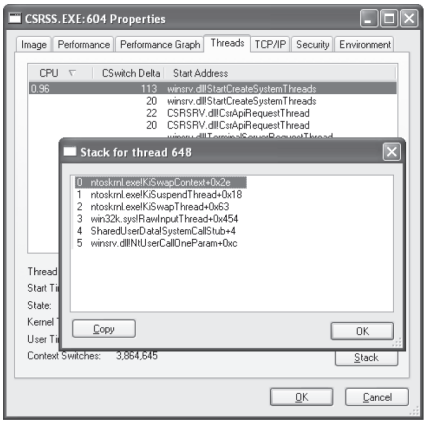
```
AttachedTo (Lower) 8a2e8ac8 \Driver\TermDD
Device queue is not busy.
```

Устройство, которому адресован данный IRP, — KeyboardClass1. Наличие объекта «устройство», принадлежащего драйверу Termdd, сообщает, что этот объект представляет ввод от клиента службы терминалов, а не с физической клавиатуры. (Листинг был получен в системе с Windows XP.)

Детальные сведения о потоке и процессе, выдавшем этот IRP, можно просмотреть командами *!thread* и *!process*:

```
kd> !thread 8a3acbc0
THREAD 8a3acbc0  Cid 025c.0288  Teb: 7ffd9000 Win32Thread:
e101fab0 WAIT: rRequest) KernelMode Alertable
    8a3cdb30  SynchronizationEvent
    8a28b4e0  SynchronizationEvent
    8a3cc908  NotificationTimer
    8a294828  SynchronizationEvent
    805453e0  NotificationEvent
    8a2e1830  SynchronizationEvent
    8a45eeb0  SynchronizationTimer
IRP List:
    89cb3928: (0006,01b4) Flags: 00000970  Mdl: 00000000
    8a1716f0: (0006,01b4) Flags: 00000970  Mdl: 00000000
Not impersonating
DeviceMap                 e10087c0 Owning Process          8a3a08b8
Wait Start TickCount      6844081
Context Switch Count      2130848                 LargeStack
UserTime                  00:00:00.0000
KernelTime                00:00:03.0274
Start Address 0x75b6e8ad
Stack Init bafa0000 Current baf9fa68 Base bafa0000 Limit baf9d000
Call 0
Priority 13 BasePriority 13 PriorityDecrement 0 DecrementCount 16

kd> !process 8a3a08b8 0
GetPointerFromAddress: unable to read from 80543ed4
PROCESS 8a3a08b8  SessionId: 0  Cid: 025c    Peb: 7ffdf000
    ParentCid: 0220   DirBase: 139af000  ObjectTable: e1c0b3d8
    HandleCount: 581. Image: CSRSS.EXE
```

Найдя этот поток в Process Explorer (*www.sysinternals.com*) на вкладке Threads окна свойств для Csrss.exe, вы убедитесь, что, судя по именам функций в его стеке, он действительно является потоком необработанного ввода (raw input thread) для подсистемы Windows.



После того как драйвер диска завершает передачу данных, диск генерирует прерывание, и ввод-вывод завершается (рис. 9-17).

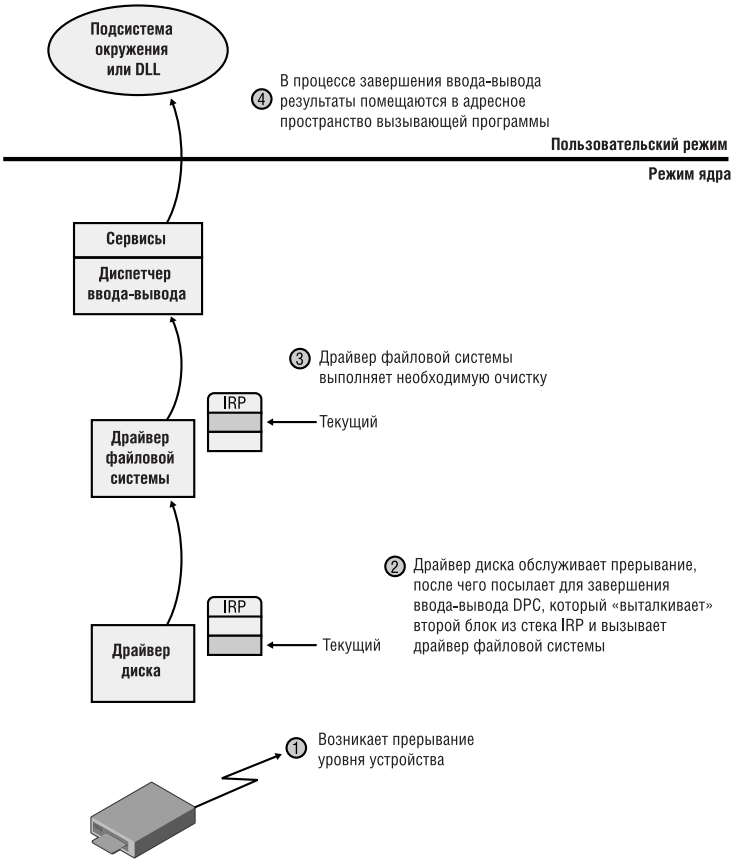


Рис. 9-17. *Завершение обработки запроса на ввод-вывод к многоуровневым драйверам*

В качестве альтернативы повторному использованию единственного IRP файловая система может создать группу *сопоставленных IRP* (associated IRPs), которые будут обрабатываться параллельно. Например, если данные, которые нужно считать из файла, разбросаны по всему диску, драйвер файловой системы может создать несколько IRP, каждый из которых инициирует чтение данных из отдельного сектора. Этот случай иллюстрирует рис. 9-18.

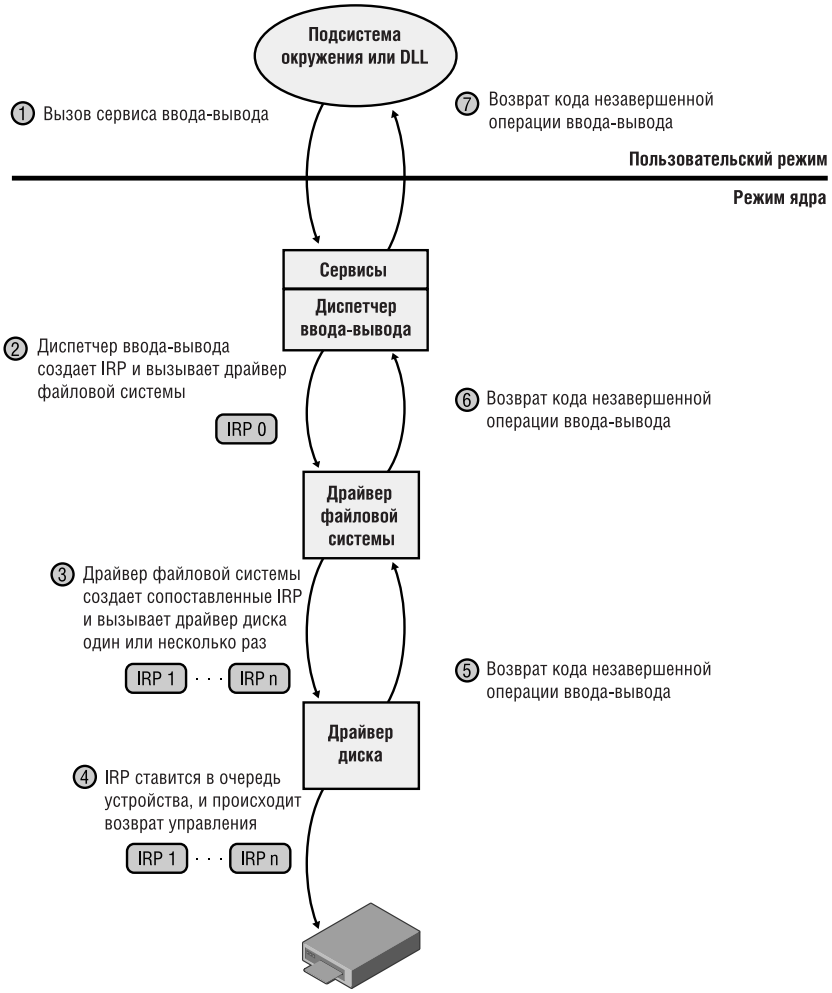


Рис. 9-18. *Обработка с использованием сопоставленных IRP*

Драйвер файловой системы передает сопоставленные IRP драйверу устройства, который ставит их в очередь устройства. Они обрабатываются по одному, а файловая система отслеживает возвращаемые данные. Когда выполнение всех сопоставленных IRP заканчивается, подсистема ввода-вывода завершает обработку исходного IRP и возвращает управление вызывающему потоку (рис. 9-19).

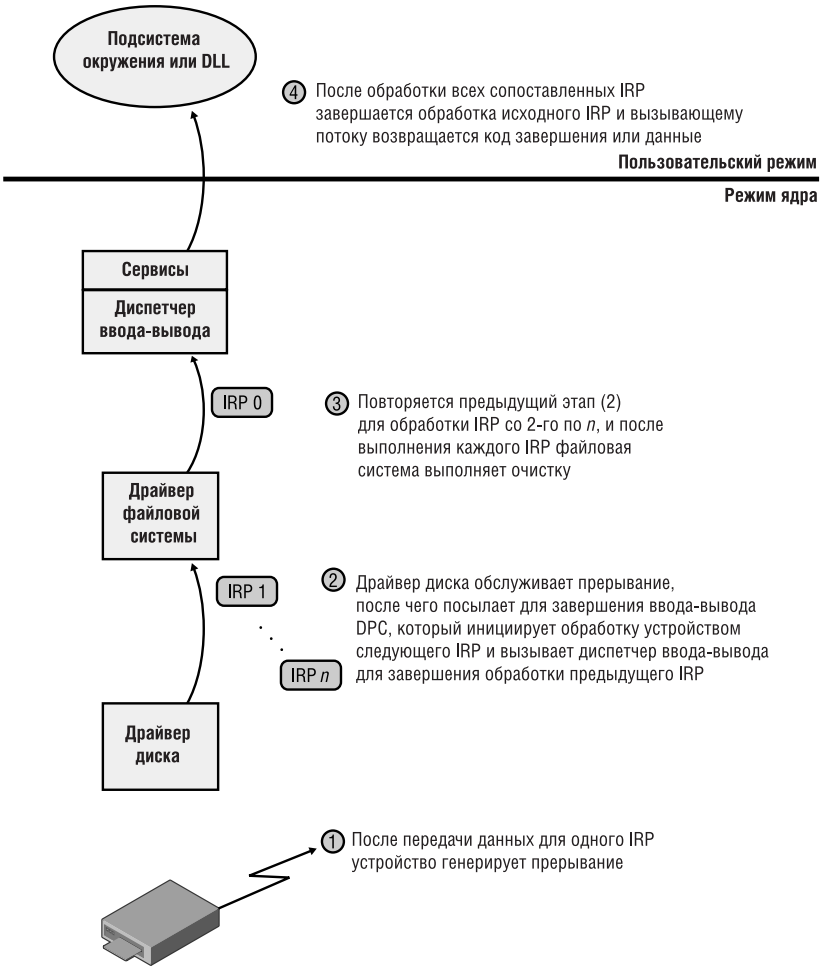


Рис. 9-19. *Завершение обработки сопоставленных IRP*

ПРИМЕЧАНИЕ Все драйверы, управляющие дисковыми файловыми системами в Windows, являются частью как минимум трехуровневого стека драйверов: драйвер файловой системы находится на верхнем уровне, диспетчер томов — на среднем, а драйвер диска — на нижнем. Кроме того, между этими драйверами может размещаться любое число драйверов фильтров. Для ясности в предыдущем примере были показаны лишь драйверы файловой системы и диска. Подробнее об управлении внешней памятью см. главу 10.

Порты завершения ввода-вывода

Создание высокопроизводительного серверного приложения требует реализации эффективной модели многопоточности. Как нехватка, так и избыток

серверных потоков, обрабатывающих клиентские запросы, приведет к проблемам с производительностью. Например, если сервер создает единственный поток для обработки всех запросов, клиенты будут «голодать», так как сервер будет обрабатывать по одному запросу единовременно. Конечно, единственный поток мог бы обрабатывать сразу несколько запросов, переключаясь с одной операции ввода-вывода на другую. Однако такая архитектура крайне сложна и не позволяет использовать преимущества многопроцессорных систем. Другая крайность — создание сервером огромного пула потоков, когда для обработки чуть ли не каждого клиентского запроса выделяется свой поток. Этот сценарий обычно ведет к перегрузке процессоров потоками: множество потоков пробуждается, выполняет обработку данных, блокируется в ожидании ввода-вывода, а после обработки запроса снова блокируется в ожидании нового запроса. Одно только наличие слишком большого количества потоков заставило бы планировщик чрезмерно часто переключать контекст, и в итоге он отобрал бы на себя немалую часть процессорного времени.

Задача сервера — свести к минимуму число переключений контекста, чтобы избежать излишнего блокирования потоков, и в то же время добиться максимального параллелизма в обработке за счет множества потоков. С этой точки зрения идеальна ситуация, при которой к каждому процессору подключен один активно обрабатывающий клиентские запросы поток, что позволяет обойтись без блокировки потоков, если на момент завершения обработки текущих запросов их ждут другие запросы. Однако такая оптимизация требует, чтобы у приложения была возможность активизировать другой поток, когда поток, обрабатывающий клиентский запрос, блокируется в ожидании ввода-вывода (например, для чтения файла в процессе обработки).

Объект *IoCompletion*

Приложения используют объект *IoCompletion* исполнительной системы, который экспортируется в Windows как *порт завершения* (completion port) — фокальная точка завершения ввода-вывода, сопоставляемая с множеством описателей файлов. Если какой-нибудь файл сопоставлен с портом завершения, то по окончании любой операции асинхронного ввода-вывода, связанной с этим файлом, в очередь порта завершения ставится пакет завершения (completion packet). Ожидание завершения любой из операций ввода-вывода в нескольких файлах может быть реализовано простым ожиданием соответствующего пакета завершения, который должен появиться в очереди порта завершения. Windows API поддерживает аналогичную функциональность через *WaitForMultipleObjects*, но порты завершения дают одно большое преимущество: число потоков, активно обслуживающих клиентские запросы, контролируется самой системой.

Создавая порт завершения, приложение указывает максимальное число сопоставленных с портом потоков, которые могут быть активны. Как уже говорилось, в идеале на каждом процессоре должно быть по одному активному потоку. Windows использует это значение для контроля числа актив-

ных потоков приложения. Если число активных потоков, сопоставленных с портом, равно заданному максимальному значению, выполнение потока, ждущего на порте завершения, запрещено. По завершении обработки текущего запроса один из активных потоков проверяет, имеется ли в очереди порта другой пакет. Если да, он просто извлекает этот пакет из очереди и переходит к обработке соответствующих данных; контекст при этом не переключается.

Использование портов завершения

Высокоуровневая схема работы порта завершения представлена на рис. 9-20. Порт завершения создается вызовом Windows-функции *CreateIoCompletionPort*. Потоки, блокированные на порте завершения, считаются сопоставленными с ним и пробуждаются по принципу LIFO («последним пришел — первым вышел»), т. е. следующий пакет достается потоку, заблокированному последним. Стеки потоков, блокируемых в течение длительного времени, могут быть выгружены в страничный файл. В итоге, если с портом сопоставлено больше потоков, чем нужно для обработки текущих заданий, система автоматически минимизирует объем памяти, занимаемой слишком долго блокируемыми потоками.

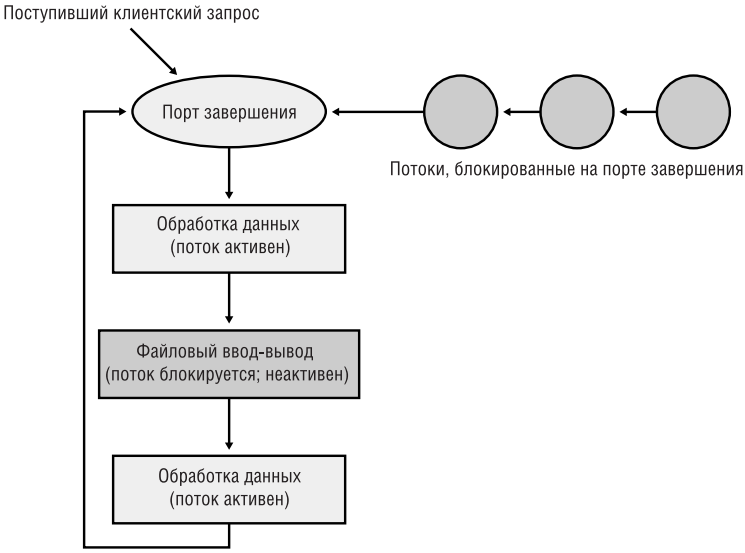


Рис. 9-20. *Так работает порт завершения ввода-вывода*

Серверное приложение обычно получает клиентские запросы через конечные точки, представляемые как описатели файлов. Пример — сокеты Windows Sockets 2 (Winsock2) или именованные каналы. Создавая конечные точки своих коммуникационных связей, сервер сопоставляет их с портом завершения, и серверные потоки ждут входящие запросы, вызывая для этого порта функцию *GetQueuedCompletionStatus*. Получив пакет из порта завер-

шения, поток начинает обработку запроса и становится активным. В процессе обработки данных поток может часто блокироваться, например из-за необходимости считать данные из файла или записать их в него, а также из-за синхронизации с другими потоками. Windows обнаруживает такие действия и выясняет, что одним активным потоком на порте завершения стало меньше. Поэтому, как только поток блокируется и становится неактивным, операционная система пробуждает другой ждущий на порте завершения поток (если в очереди есть пакет).

Microsoft рекомендует устанавливать максимальное число активных потоков на порте завершения примерно равным числу процессоров в системе. Имейте в виду, что это значение может быть превышено. Допустим, вы задали, что максимальное значение должно быть равно 1. При поступлении клиентского запроса выделенный для его обработки поток становится активным. Поступает второй запрос, но второй поток не может продолжить его обработку, так как лимит уже достигнут. Затем первый поток блокируется в ожидании файлового ввода-вывода и становится неактивным. Тогда освобождается второй поток и, пока он активен, завершается файловый ввод-вывод для первого потока, в результате чего первый поток вновь активизируется. С этого момента и до блокировки одного из потоков число активных потоков превышает установленный лимит на 1.

API, предусмотренный для порта завершения, также позволяет серверному приложению ставить в очередь порта завершения самостоятельно определенные пакеты завершения; для этого предназначена функция *PostQueuedCompletionStatus*. Сервер обычно использует эту функцию для уведомления своих потоков о внешних событиях, например о необходимости корректного завершения работы.

Как работает порт завершения ввода-вывода

Windows-приложения создают порты завершения вызовом Windows-функции *CreateIoCompletionPort* с указанием NULL вместо описателя порта завершения. Это приводит к выполнению системного сервиса *NtCreateIoCompletion*. Объект *IoCompletion* исполнительной системы, построенный на основе синхронизующего объекта ядра, называется *очередью*. Таким образом, системный сервис создает объект «порт завершения» и инициализирует объект «очередь» в памяти, выделенной для порта. (Указатель на порт ссылается и на объект «очередь», так как последний находится в начальной области памяти порта.) Максимальное число сопоставленных с портом потоков, которые могут быть активны, указывается в объекте «очередь» при его инициализации; это значение, которое было передано в *CreateIoCompletionPort*. Для инициализации объекта «очередь» порта завершения *NtCreateIoCompletion* вызывает функцию *KeInitializeQueue*.

Когда приложение обращается к *CreateIoCompletionPort* для связывания описателя файла с портом, вызывается системный сервис *NtSetInformationFile*, которому передается описатель этого файла. При этом класс информации для *NtSetInformationFile* устанавливается как *FileCompletionInformation*, и,

кроме того, эта функция принимает описатель порта завершения и параметр *CompletionKey*, ранее переданный в *CreateIoCompletionPort*. Функция *NtSetInformationFile* производит разыменование описателя файла для получения объекта «файл» и создает структуру данных контекста завершения.

Указатель на эту структуру *NtSetInformationFile* помещает в поле *CompletionContext* объекта «файл». По завершении асинхронной операции ввода-вывода для объекта «файл» диспетчер ввода-вывода проверяет, отличается ли поле *CompletionContext* от NULL. Если да, он создает пакет завершения и ставит его в очередь порта завершения вызовом *KeInsertQueue*; при этом в качестве очереди, в которую помещается пакет, указывается порт. (Здесь объект «порт завершения» — синоним объекта «очередь».)

Когда серверный поток вызывает *GetQueuedCompletionStatus*, выполняется системный сервис *NtRemoveIoCompletion*. После проверки параметров и преобразования описателя порта завершения в указатель на порт *NtRemoveIoCompletion* вызывает *KeRemoveQueue*.

Как видите, *KeRemoveQueue* и *KeInsertQueue* — это базовые функции, обеспечивающие работу порта завершения. Они определяют, следует ли активизировать поток, ждущий пакет завершения ввода-вывода. Объект «очередь» поддерживает внутренний счетчик активных потоков и хранит такое значение, как максимальное число активных потоков. Если при вызове потоком *KeRemoveQueue* текущее число активных потоков равно максимуму или превышает его, данный поток будет включен (в порядке LIFO) в список потоков, ждущих пакет завершения. Список потоков отделен от объекта «очередь». В блоке управления потоком имеется поле для указателя на очередь, сопоставленную с объектом «очередь»; если это поле пустое, поток не связан с очередью.

Windows отслеживает потоки, ставшие неактивными из-за ожидания на каких-либо объектах, отличных от порта завершения, по указателю на очередь, присутствующему в блоке управления потоком. Процедуры планировщика, в результате выполнения которых поток может быть блокирован (*KeWaitForSingleObject*, *KeDelayExecutionThread* и т. д.), проверяют этот указатель. Если он не равен NULL, они вызывают функцию *KiActivateWaiterQueue*, которая уменьшает счетчик числа активных потоков, сопоставленных с очередью. Если конечное число меньше максимального и в очереди есть хотя бы один пакет завершения, первый поток из списка потоков очереди пробуждается и получает самый старый пакет. И напротив, всякий раз, когда после блокировки пробуждается поток, связанный с очередью, планировщик выполняет функцию *KiUnwaitThread*, увеличивающую счетчик числа активных потоков очереди.

Наконец, в результате вызова Windows-функции *PostQueuedCompletionStatus* выполняется системный сервис *NtSetIoCompletion*, который просто вставляет с помощью *KeInsertQueue* специальный пакет в очередь порта завершения.

Порт завершения в действии показан на рис. 9-21. Хотя к обработке пакетов завершения готовы два потока, максимум, равный 1, допускает активизацию только одного потока, связанного с портом завершения. Таким образом, на этом порте завершения блокируется два потока.

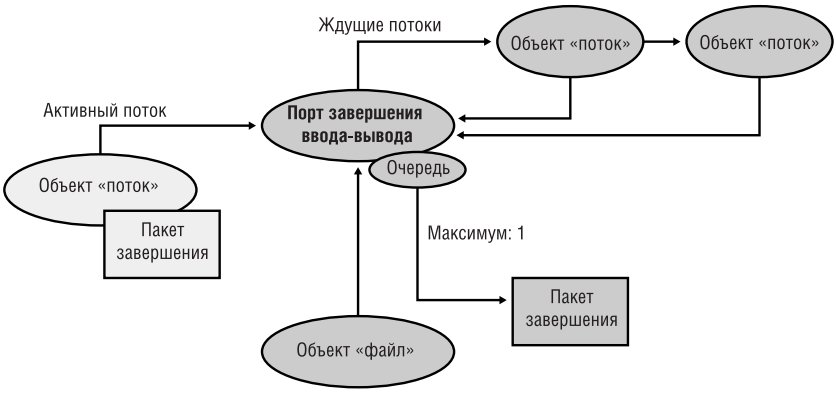


Рис. 9-21. *Порт завершения ввода-вывода в действии*

Driver Verifier

Утилита Driver Verifier (о которой мы уже рассказывали в главе 7) предоставляет несколько параметров для проверки правильности операций, связанных с вводом-выводом. На рис. 9-22 в окне Driver Verifier Manager (Диспетчер проверки драйверов) в Windows Server 2003 эти параметры помечены флажками.

Даже если вы не указываете никаких параметров, Verifier наблюдает за работой выбранных для верификации драйверов, следя за недопустимыми операциями, в том числе за вызовом функций пула памяти ядра при неправильном уровне IRQL, попытками повторного освобождения свободной памяти и запроса блоков памяти нулевого размера.

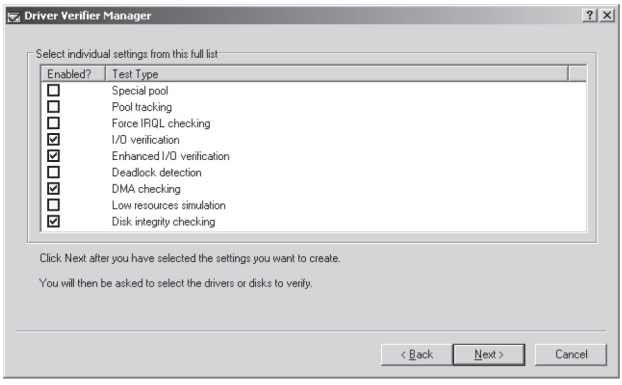


Рис. 9-22. *Параметры Driver Verifier, относящиеся к операциям ввода-вывода*

Параметры проверки ввода-вывода перечислены ниже.

- **I/O Verification (Проверка ввода-вывода)** Если этот параметр выбран, диспетчер ввода-вывода выделяет память под IRP-пакеты для проверяемых драйверов из специального пула и отслеживает его использова-

ние. Кроме того, Verifier вызывает крах системы по окончании обработки IRP с неправильным состоянием и при передаче неверного объекта «устройство» диспетчеру ввода-вывода. (В Windows 2000 этот параметр назывался I/O Verification Level 1).

- **I/O Verification Level 2 (Проверка ввода-вывода уровня 2)** Этот параметр существует только в Windows 2000; он просто ужесточает проверку операций обработки IRP и использования стека.
- **Enhanced I/O Verification (Расширенная проверка ввода-вывода)** Этот параметр впервые появился в Windows XP и включает мониторинг всех IRP для контроля того, что драйверы корректно помечают их при асинхронной обработке, что они правильно управляют блоками стека устройства и что они удаляют каждый объект «устройство» только раз. В дополнение Verifier случайным образом посылает драйверам ложные IRP, связанные с управлением электропитанием и WMI, изменяет порядок перечисления устройств и изменяет состояние IRP, связанных с PnP и электропитанием, по окончании их обработки; последнее позволяет выявить драйверы, возвращающие неверное состояние из своих процедур диспетчеризации.
- **DMA Checking (Проверка DMA)** DMA — аппаратно поддерживаемый механизм, позволяющий устройствам передавать данные в физическую память или получать их из нее без участия процессора. Диспетчер ввода-вывода поддерживает ряд функций, используемых драйверами для планирования DMA-операций и управления ими. Данный параметр включает проверку правильности применения этих функций и буферов, предоставляемых диспетчером ввода-вывода для DMA-операций.
- **Disk Integrity Verification (Проверка целостности диска)** После включения этого параметра, доступного только в Windows Server 2003, Verifier ведет мониторинг операций чтения и записи на дисках и проверяет контрольные суммы соответствующих данных. По окончании операций чтения с диска Verifier проверяет ранее сохраненные контрольные суммы и вызывает крах системы, если новая и старая контрольные суммы не совпадают, так как это свидетельствует о повреждении диска на аппаратном уровне.
- **SCSI Verification (Проверка SCSI)** Этот параметр появился в Windows XP и не виден в диалоговом окне параметров Driver Verifier. Однако он включается, когда вы выбираете для проверки минипорт-драйвер SCSI и отмечаете хотя бы один из других параметров. Тогда Verifier следит, как минипорт-драйвер SCSI использует функции, предоставляемые драйвером библиотеки SCSI-минипорта — storport.sys или scsiport.sys. При этом проверяется, что драйвер не обрабатывает запрос более одного раза, что он не передает недопустимые аргументы и что на выполнение операций не уходит больше определенного времени. (Подробнее о минипорт-драйверах SCSI см. в главе 10.)

Driver Verifier предназначен главным образом разработчикам драйверов устройств и помогает им обнаруживать ошибки в своем коде. Однако это еще и мощный инструмент для системных администраторов, позволяющий анализировать причины краха. Подробнее о роли Driver Verifier в анализе краха системы см. в главе 14.

Диспетчер Plug and Play (PnP)

Диспетчер PnP — основной компонент, от которого зависит способность Windows к распознаванию изменений в аппаратной конфигурации. Благодаря этому от пользователя не требуется знания тонкостей настройки устройств и системы при их установке и удалении. Так, диспетчер PnP позволяет портативному компьютеру с Windows при подключении к стыковочной станции автоматически обнаруживать дополнительные устройства стыковочной станции и делать их доступными пользователю.

Поддержка Plug and Play требует взаимодействия на уровнях оборудования, драйверов устройств и операционной системы. Эта поддержка в Windows базируется на промышленных стандартах перечисления и идентификации подключенных к шинам устройств. Например, стандарт USB определяет способ самоидентификации устройств, подключенных к шине USB. На этой основе в Windows реализуются следующие возможности Plug and Play.

- Диспетчер PnP автоматически распознает установленные устройства, и этот процесс включает перечисление устройств при загрузке и обнаружение их добавления или удаления во время работы системы.
- Диспетчер PnP выделяет аппаратные ресурсы, собирая информацию о требованиях устройств к аппаратным ресурсам (прерывания, диапазоны адресов ввода-вывода, регистры ввода-вывода или ресурсы, специфичные для шин). В ходе *арбитража ресурсов* (resource arbitration) диспетчер PnP распределяет ресурсы между устройствами с учетом их требований. Поскольку устройства могут быть добавлены в систему после распределения ресурсов на этапе загрузки, диспетчер PnP должен уметь перераспределять ресурсы.
- Другая функция диспетчера PnP — загрузка соответствующих драйверов. На основе идентификационных данных устройства он определяет, установлен ли в системе драйвер, способный управлять этим устройством. Если да, диспетчер PnP указывает диспетчеру ввода-вывода загрузить его. Если подходящий драйвер не установлен, диспетчер PnP режима ядра взаимодействует с диспетчером PnP пользовательского режима, чтобы установить устройство. При этом он может попросить пользователя указать местонахождение нужных драйверов.
- Диспетчер PnP также реализует механизмы, позволяющие приложениям и драйверам обнаруживать изменения в аппаратной конфигурации. Иногда для работы драйверов и приложений требуется определенное устройство, поэтому в Windows имеются средства, которые дают возмож-

ность таким драйверам и приложениям запрашивать уведомления о наличии, добавлении и удалении устройств.

Уровень поддержки Plug and Play

Windows нацелена на полную поддержку Plug and Play, но конкретный уровень поддержки зависит от устройств, подключенных к системе, и установленных в ней драйверов. Уровень поддержки Plug and Play может быть снижен, если хотя бы один драйвер или устройство не отвечает стандарту Plug and Play. Более того, драйвер, не поддерживающий Plug and Play, может лишить систему возможности использовать другие устройства. В таблице 9-2 показано, к каким результатам приводят различные сочетания устройств и драйверов с поддержкой Plug and Play и без нее.

Таблица 9-2. *Возможности устройств и драйверов в поддержке Plug and Play*

Устройство	Драйвер	
	PnP-совместимый	PnP-несовместимый
PnP-совместимое	Полная поддержка PnP	Поддержка PnP невозможна
PnP-несовместимое	Частичная поддержка PnP	Поддержка PnP невозможна

PnP-несовместимое устройство, например унаследованная звуковая плата с ISA-шиной, не поддерживает автоматическое определение. Из-за этого таким устройствам запрещены некоторые операции вроде «горячего» подключения или перехода в один из режимов сна. Если для такого устройства вручную установить PnP-совместимый драйвер, он сможет по крайней мере использовать ресурсы, которые диспетчер PnP будет выделять этому устройству.

Унаследованные драйверы, например драйверы, разработанные для Windows NT 4, не совместимы с Plug and Play. Хотя они работают в Windows, диспетчер PnP не сможет динамически перераспределять ресурсы, назначенные таким устройствам. Допустим, унаследованное устройство использует для ввода-вывода диапазон памяти A или B. При загрузке системы диспетчер PnP выделяет этому устройству диапазон A. Если впоследствии в систему будет добавлено устройство, способное использовать только диапазон A, диспетчер PnP не сможет указать драйверу первого устройства перенастроить его на диапазон B. Из-за этого второе устройство не получит нужные ресурсы и будет недоступно. Унаследованные драйверы также мешают переходу системы в один из режимов сна (см. раздел «Диспетчер электропитания» далее в этой главе).

Поддержка Plug and Play со стороны драйвера

Для поддержки Plug and Play в драйвере должна быть реализована процедура диспетчеризации Plug and Play, а также процедура добавления устройства. Однако драйверы шин должны поддерживать типы запросов Plug and Play, отличные от тех, которые поддерживаются функциональными драйверами и драйверами фильтров. Так, при перечислении устройств в процессе загрузки диспетчер PnP запрашивает у драйверов шин описание устройств, най-

денных ими на своих шинах. В это описание входят данные, уникально идентифицирующие каждое устройство, а также требования устройств к аппаратным ресурсам. Диспетчер PnP принимает эту информацию и загружает функциональные драйверы или драйверы фильтров, установленные для обнаруженных устройств. Затем он вызывает процедуру добавления устройства каждого драйвера, установленного для каждого устройства.

Выполняя процедуру добавления устройства, функциональные драйверы и драйверы фильтров готовятся начать управление своими устройствами, но на самом деле пока еще не взаимодействуют с ними. Они ждут команду *start-device*, которую диспетчер PnP должен передать их процедурам диспетчеризации Plug and Play. До передачи этой команды диспетчер PnP выполняет арбитраж ресурсов, чтобы решить, какие ресурсы выделить тому или иному устройству. В команде *start-device* указываются назначенные ресурсы, определенные диспетчером PnP при арбитраже ресурсов. Получив команду *start-device*, драйвер может настроить свое устройство на использование указанных ресурсов. Если программа пытается открыть устройство, которое не готово к началу работы, она получает код ошибки, указывающий на отсутствие этого устройства.

После запуска устройства диспетчер PnP может посылать драйверу дополнительные PnP-команды, в том числе относящиеся к удалению устройства из системы или перераспределению ресурсов. Например, когда пользователь запускает утилиту, показанную на рис. 9-23, — для ее запуска надо щелкнуть правой кнопкой мыши значок платы PC Card на панели задач и выбрать команду Unplug Or Eject Hardware (Отключение или извлечение аппаратного устройства), — и командует Windows извлечь PCMCIA-плату, диспетчер PnP посылает уведомление *query-remove* каждому приложению, зарегистрированному на получение PnP-уведомлений об этом устройстве. Как правило, приложения регистрируются на получение уведомлений через свои описатели устройства, которые они закрывают, получая уведомление *query-remove*. Если ни одно приложение не налагает вето на запрос *query-remove*, диспетчер PnP посылает команду *query-remove* драйверу, управляющему извлекаемым устройством. На этом этапе драйвер решает, что ему делать дальше: запретить удаление устройства или завершить все операции ввода-вывода на этом устройстве и прекратить дальнейший прием запросов на ввод-вывод, направляемых устройству. Если драйвер отвечает согласием на запрос об удалении и открытых описателей устройства больше нет, диспетчер PnP посылает драйверу команду *remove*, требующую от него прекратить обращение к устройству и освободить все ресурсы, выделенные им для данного устройства.

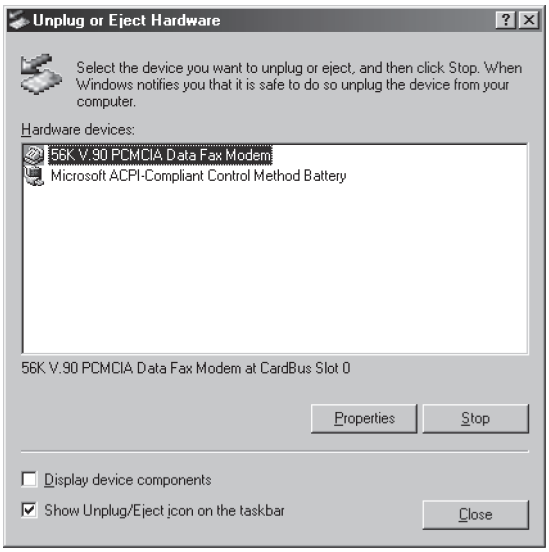


Рис. 9-23. *Утилита для отключения или извлечения платы PC Card*

Когда диспетчеру PnP нужно перераспределить ресурсы для устройства, он сначала запрашивает драйвер, может ли тот временно приостановить операции на устройстве, и с этой целью посылает команду *query-stop*. Драйвер отвечает на этот запрос согласием, если нет риска потери или повреждения данных; в ином случае он отклоняет такой запрос. Как и в случае команды *query-remove*, драйвер, согласившись с запросом, заканчивает незавершенные операции ввода-вывода и больше не передает этому устройству запросы на ввод-вывод. (Новые запросы на ввод-вывод драйвер обычно ставит в очередь.) Далее диспетчер PnP посылает драйверу команду *stop*. На этом этапе диспетчер PnP может указать драйверу выделить устройству другие ресурсы, а потом послать команду *start-device*.

Команды Plug and Play вызывают переход устройства в строго определенные состояния, которые в упрощенной форме представлены на рис. 9-24. (Некоторые состояния и команды Plug and Play на этой иллюстрации опущены. Кроме того, этот вариант относится к диаграмме состояний, реализуемой функциональными драйверами. Диаграмма состояний, реализуемых драйверами шин, гораздо сложнее.) Кстати, на рис. 9-24 показано одно из состояний, которое мы еще не обсудили, — устройство переходит в него после команды *surprise-remove* диспетчера PnP. Эта команда посылается при неожиданном удалении устройства из системы, например из-за его отказа или из-за извлечения PCMCIA-платы без применения соответствующей утилиты. Команда *surprise-remove* заставляет драйвер немедленно прекратить всякое взаимодействие с устройством, так как оно больше не подключено к системе, и отменить любые незавершенные запросы ввода-вывода.

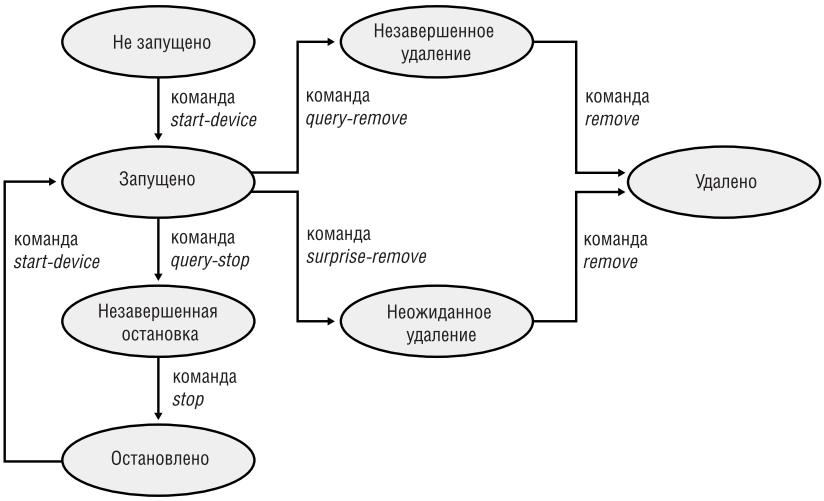


Рис. 9-24. *Переходы PnP-состояний устройств*

Загрузка, инициализация и установка драйвера

Драйвер может загружаться в Windows явно и на основе перечисления. Явную загрузку определяет ветвь реестра HKLM\SYSTEM\CurrentControlSet\Services, и на эту тему см. раздел «Сервисные приложения» главы 4. Загрузка на основе перечисления происходит при динамической загрузке диспетчером PnP драйверов для устройств, о наличии которых сообщает драйвер шины.

Параметр Start

В главе 4 мы объяснили, что у каждого драйвера и Windows-сервиса есть свой раздел в ветви реестра Services текущего набора параметров управления. В этот раздел входят параметры, указывающие тип образа (например, Windows-сервис, драйвер или файловая система), путь к файлу образа драйвера или сервиса и параметры, контролирующие порядок загрузки драйвера или сервиса. Между загрузкой Windows-сервисов и явной загрузкой драйверов есть два главных различия:

- только для драйверов устройств в параметре Start могут быть указаны значения 0 (запуск при загрузке системы) и 1 (запуск системой);
- драйверы устройств могут использовать параметры Group и Tag для контроля порядка своей загрузки при запуске системы, но в отличие от сервисов не могут определять параметры DependOnGroup или DependOnService.

В главе 5 мы рассмотрели этапы процесса загрузки и объяснили, что параметр Start драйвера, равный 0, означает, что этот драйвер загружается загрузчиком операционной системы. А если Start равен 1, драйвер загружается диспетчером ввода-вывода после инициализации компонентов исполнительной системы. Диспетчер ввода-вывода вызывает инициализирующие процедуры драйверов в том порядке, в каком драйверы загружались при запуске системы. Как и Windows-сервисы, драйверы используют параметр

Group в своем разделе реестра, чтобы указать группу, к которой они принадлежат; порядок загрузки групп определяется параметром HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder\List.

Драйвер может еще больше детализировать порядок своей загрузки с помощью параметра Tag, который указывает конкретную позицию драйвера в группе. Диспетчер ввода-вывода сортирует драйверы в группе по значениям параметров Tag, определенных в разделах реестра, соответствующих этим драйверам. Драйверы, не имеющие параметра Tag, перемещаются в конец списка драйверов группы. Вы могли предположить, что диспетчер ввода-вывода сначала инициализирует драйверы с меньшими значениями Tag, потом — с бльшими, но это не так. Приоритет значений параметров Tag в рамках группы определяется в HKLM\SYSTEM\CurrentControlSet\Control\GroupOrderList; этот раздел реестра дает Microsoft и разработчикам драйверов свободу в определении собственной системы целых чисел.

Вот правила, по которым драйверы устанавливают значение своего параметра Start.

- Драйверы, не поддерживающие Plug and Play, настраивают Start так, чтобы система загружала их на определенном этапе своего запуска.
- Драйверы, которые должны загружаться системным загрузчиком при запуске операционной системы, указывают в Start значение 0 (запуск при загрузке системы). Пример — драйверы системных шин и драйвер файловой системы, используемый при загрузке системы.
- Драйвер, который не требуется для загрузки системы и распознает устройство, не перечисляемое драйвером системной шины, указывает в Start значение, равное 1 (запуск системой). Пример — драйвер последовательного порта, информирующий диспетчер PnP о присутствии стандартных последовательных портов, которые были обнаружены программой Setup и зарегистрированы в реестре.
- Драйвер, не поддерживающий Plug and Play, или драйвер файловой системы, не обязательный для загрузки системы, устанавливает значение Start равным 2 (автозапуск). Пример — драйвер многосетевого UNC-провайдера (Multiple UNC Provider, MUP), поддерживающий UNC-имена удаленных ресурсов (вроде \\REMOTECOMPUTERNAME\SHARE).
- PnP-драйверы, не нужные для загрузки системы (например, драйверы сетевых адаптеров), указывают значение Start равным 3 (запуск по требованию).

Единственное предназначение параметра Start для PnP-драйверов и драйверов перечисляемых устройств — загрузка драйвера с помощью загрузчика операционной системы, если такой драйвер обязателен для успешного запуска системы.

Перечисление устройств

Диспетчер PnP начинает перечисление устройств с виртуального драйвера шины с именем Root, который представляет всю систему и выступает в роли драйвера шины для драйверов, не поддерживающих Plug and Play, и для HAL.

HAL работает как драйвер шины, перечисляющий устройства, напрямую подключенные к материнской плате, и такие системные компоненты, как аккумуляторы. Определяя основную шину (обычно это PCI-шина) и устройства типа аккумуляторов и вентиляторов, HAL на самом деле полагается на описание оборудования, зафиксированное программой Setup в реестре еще при установке операционной системы.

Драйвер основной шины перечисляет устройства на этой шине, при этом он может найти другие шины, драйверы которых инициализируются диспетчером PnP. Эти драйверы в свою очередь могут обнаруживать другие устройства, включая вспомогательные шины. Такой рекурсивный процесс — перечисление, загрузка драйвера (если он еще не загружен), дальнейшее перечисление — продолжается до тех пор, пока не будут обнаружены и сконфигурированы все устройства в системе.

По мере поступления сообщений от драйверов шин об обнаруженных устройствах, диспетчер PnP формирует внутреннее дерево, называемое *деревом устройств* (device tree) и отражающее взаимосвязи между устройствами. Узлы этого дерева называются *узлами устройств* (device nodes, devnodes). Узел устройств содержит информацию об объектах «устройство», представляющих устройства, и другую PnP-информацию, которая записывается в узел диспетчером PnP. Упрощенный пример дерева устройств показан на рис. 9-25. Эта система ACPI-совместима, и поэтому перечислителем основной шины является ACPI-совместимый HAL. К основной шине PCI в данной системе подключены шины USB, ISA и SCSI.

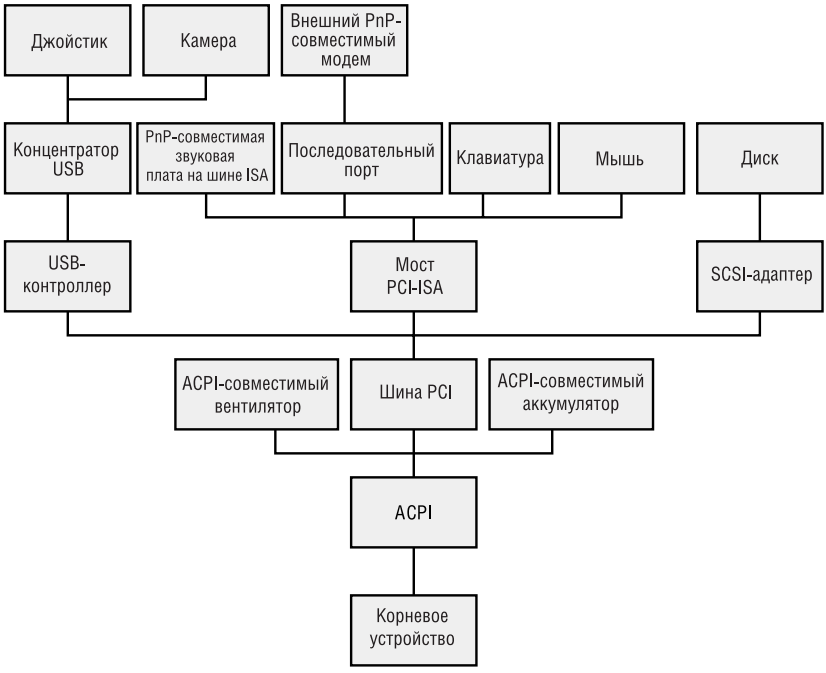


Рис. 9-25. *Пример дерева устройств*

Диспетчер устройств, доступный из оснастки Computer Management (Управление компьютером) и с вкладки Hardware (Оборудование) окна свойств системы, отображает простой список устройств в системе, сконфигурированной по умолчанию. Для просмотра устройств в виде иерархического дерева можно выбрать в меню View (Вид) диспетчера устройств команду Devices By Connection (Устройства по подключению). Рис. 9-26 иллюстрирует, как выглядит окно диспетчера устройств при выборе этой команды.

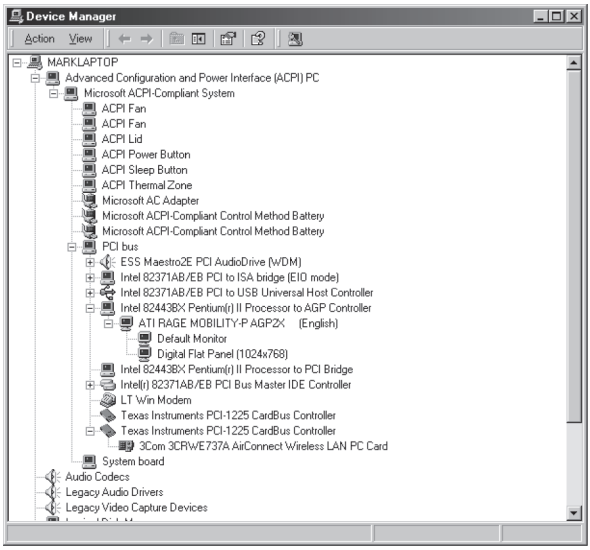


Рис. 9-26. *Диспетчер устройств, показывающий дерево устройств*

С учетом перечисления устройств загрузка и инициализация драйверов происходит в следующем порядке.

1. Диспетчер ввода-вывода вызывает входную процедуру каждого драйвера, запускаемого при загрузке системы. Если у такого драйвера имеются дочерние устройства, диспетчер ввода-вывода перечисляет эти устройства, сообщая о них диспетчеру PnP. Дочерние устройства конфигурируются и запускаются, если их драйверы являются запускаемыми при загрузке системы. Если у устройства есть драйвер, не запускаемый при загрузке системы, диспетчер PnP создает для этого устройства узел, но не запускает устройство и не загружает его драйвер.
2. После инициализации драйверов, запускаемых при загрузке системы, диспетчер PnP проходит по дереву устройств, загружая драйверы для узлов устройств, не загруженных на первом этапе, и запускает их устройства. Запуская каждое устройство, диспетчер PnP перечисляет его дочерние устройства (если таковые есть). Для этого он запускает соответствующие драйверы и при необходимости перечисляет их дочерние устройства. На данном этапе диспетчер PnP загружает драйверы для обнаруженных устройств *независимо от значений параметров Start этих драйве-*

ров (кроме тех драйверов, параметр Start которых содержит значение «отключен»). В конце этого этапа драйверы всех PnP-устройств загружены и запущены, кроме драйверов не перечисляемых устройств и их дочерних устройств.

3. Диспетчер PnP загружает любые еще не загруженные драйверы, запускаемые системой. Эти драйверы определяют свои устройства, не перечисляемые обычным образом, и сообщают о них. После этого диспетчер PnP загружает драйверы для этих устройств.
4. Наконец, диспетчер управления сервисами (SCM) загружает автоматически запускаемые драйверы.

Дерево устройств используется диспетчерами PnP и электропитания в то время, когда они выдают устройствам IRP-пакеты, связанные с Plug and Play и управлением электропитанием. Как правило, поток IRP распространяется от верхней части узла устройств вниз, и иногда какой-либо драйвер в одном из узлов устройств создает новые IRP для передачи другим узлам (всегда в направлении корня). О потоках IRP-пакетов, связанных с Plug and Play и управлением электропитанием, мы поговорим позже.

ЭКСПЕРИМЕНТ: получаем дамп дерева устройств

Команда *!devnode* отладчика ядра позволяет более подробно изучить дерево устройств. Задав *0* и *1* в качестве параметров, вы получите дамп внутренних структур узлов этого дерева. При этом элементы структур выводятся с отступами, отражающими позиции элементов в общей иерархии.

```
lkd> !devnode 0 1
Dumping IopRootDeviceNode (= 0x8a4b7ee8)
DevNode 0x8a4b7ee8 for PDO 0x8a4b7020
  InstancePath is "HTREE\ROOT\0"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
  DevNode 0x8a4b7a50 for PDO 0x8a4b7b98
    InstancePath is "Root\ACPI_HAL\0000"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeEnumerateCompletion (0x30d)
    DevNode 0x8a4af448 for PDO 0x8a4eb2c8
      InstancePath is "ACPI_HAL\PNP0C08\0"
      ServiceName is "ACPI"
      State = DeviceNodeStarted (0x308)
      Previous State = DeviceNodeEnumerateCompletion (0x30d)
      DevNode 0x8a4af198 for PDO 0x8a4b1350
        InstancePath is "ACPI\GenuineIntel_-_x86_Family_6_
        Model_9\_0"
        ServiceName is "gv3"
        State = DeviceNodeStarted (0x308)
        Previous State = DeviceNodeEnumerateCompletion (0x30d)
```

```
      DevNode 0x8a4e8008 for PDO 0x8a4a8950
        InstancePath is "ACPI\ThermalZone\THM_"
        State = DeviceNodeStarted (0x308)
        Previous State = DeviceNodeEnumerateCompletion (0x30d)
      DevNode 0x8a4e82b8 for PDO 0x8a4eb640
        InstancePath is "ACPI\ACPI0003\2&daba3ff&0"
        ServiceName is "CmBatt"
...
```

Информация, показываемая для каждого узла устройств, включает InstancePath (имя подраздела для перечисленного устройства в HKLM\SYSTEM\CurrentControlSet\Enum) и ServiceName (имя подраздела для драйвера устройства в HKLM\SYSTEM\CurrentControlSet\Services). Чтобы просмотреть такие ресурсы, как прерывания, порты и диапазоны памяти, назначенные каждому узлу устройств, используйте команду *!devnode 0 3*.

Все устройства, обнаруженные после установки системы, регистрируются в подразделах раздела реестра HKLM\SYSTEM\CurrentControlSet\Enum. Этим подразделам присваиваются имена в виде *<Перечислитель>\<ID_устройства>\<ID_экземпляра>*, где *перечислитель* — драйвер шины, *ID_устройства* — уникальный идентификатор устройств данного типа, а *ID_экземпляра* — уникальный идентификатор данного экземпляра этого устройства.

Узлы устройств

Узел устройств, который включает минимум два объекта «устройство», показан на рис. 9-27.

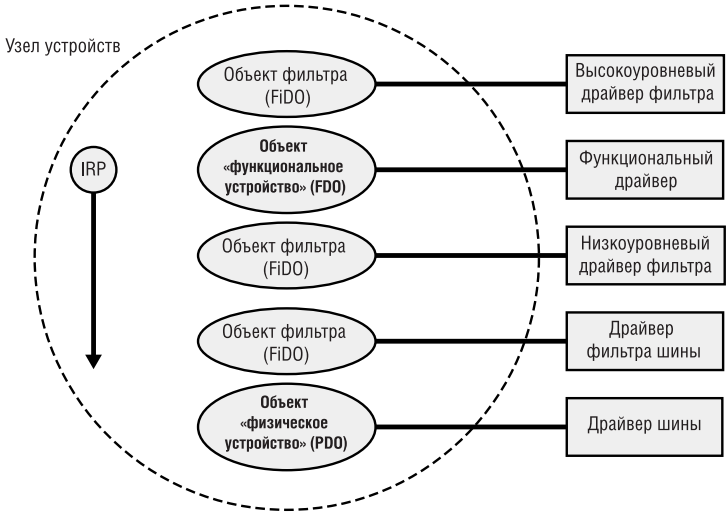


Рис. 9-27. *Внутренняя структура узла устройств*

- **Объект «физическое устройство» (physical device object, PDO)** Создается драйвером шины по заданию диспетчера PnP, когда драйвер шины, перечисляя устройства на своей шине, сообщает о наличии какого-либо устройства. PDO представляет физический интерфейс устройства.
- **Необязательные группы объектов-фильтров (filter device objects, FiDO)** Одна группа таких объектов размещается между PDO и FDO (создается драйверами фильтров шин), вторая — между FDO и первой группой FiDO (создается низкоуровневыми драйверами фильтров), третья — над FDO (создается высокоуровневыми драйверами фильтров).
- **Объект «функциональное устройство» (functional device object, FDO)** Создается функциональным драйвером, который загружается диспетчером PnP для управления обнаруженным устройством. FDO представляет логический интерфейс устройства. Функциональный драйвер может выступать и в роли драйвера шины, если к устройству, представленному FDO, подключены другие устройства. Этот драйвер часто создает интерфейс к PDO, соответствующему данному FDO, что позволяет приложениям и другим драйверам открывать устройство и взаимодействовать с ним. Иногда функциональные драйверы подразделяются на драйвер класса, порт-драйвер и минипорт-драйвер, совместно управляющие вводом-выводом для FDO.

Узлы устройств полагаются на функциональность диспетчера ввода-вывода; поток IRP идет по узлу устройств сверху вниз. Решение об окончании обработки IRP может быть принято на любом уровне узла устройств. Например, функциональный драйвер может обработать запрос на чтение, не пересылая IRP драйверу шины. IRP проходит весь путь сверху вниз и далее к узлу устройств, содержащему драйвер шины, только если функциональному драйверу нужна помощь драйвера шины.

Загрузка драйверов для узла устройств

До сих пор мы так и не ответили на два важных вопроса: как диспетчер PnP определяет, какой функциональный драйвер нужно загрузить для данного устройства и как драйверы фильтров регистрируют свое присутствие, чтобы их можно было загружать при создании узла устройств?

Ответ на оба вопроса надо искать в реестре. Перечисляя устройства, драйвер шины сообщает диспетчеру PnP идентификаторы обнаруженных устройств. Эти идентификаторы специфичны для конкретной шины. Например, для шины USB такой идентификатор состоит из идентификатора изготовителя (vendor ID, VID) и идентификатора продукта (product ID, PID), назначенного устройству изготовителем (подробнее о форматах идентификаторов устройств см. в DDK). В совокупности эти идентификаторы образуют то, что в терминологии спецификации Plug and Play называется *идентификатором устройства* (device ID). Диспетчер PnP также запрашивает у драйвера шины *идентификатор экземпляра* (instance ID), который позволяет различать отдельные экземпляры одного и того же устройства. Идентифи-

катор экземпляра может определять устройство относительно шины (например, USB-порт) или представлять глобально уникальный дескриптор (скажем, серийный номер устройства). Идентификаторы устройства и экземпляра дают *идентификатор экземпляра устройства* (device instance ID, DIID), используемый диспетчером PnP для поиска раздела устройства в ветви реестра HKLM\SYSTEM\CurrentControlSet\Enum. Пример такого раздела для клавиатуры показан на рис. 9-28. В эти разделы помещаются данные, характеризующие устройство, и получаемые из INF-файла параметры Service и ClassGUID, с помощью которых диспетчер PnP находит драйверы, нужные для данного устройства.

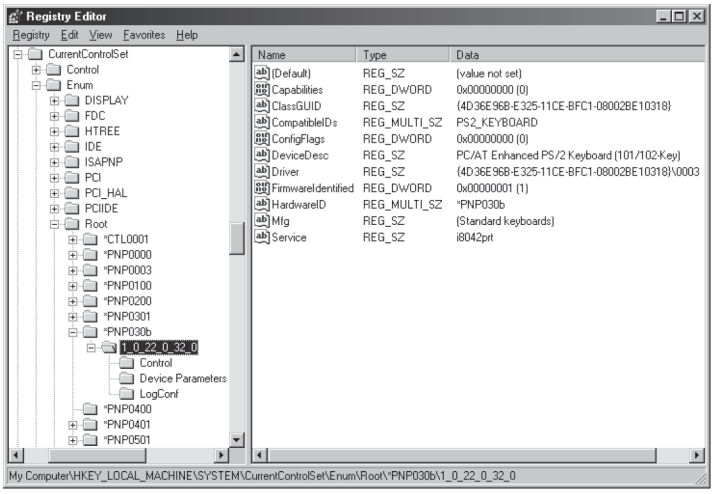


Рис. 9-28. *Раздел реестра для клавиатуры, создаваемый в процессе перечисления*

ЭКСПЕРИМЕНТ: просмотр детальных сведений об узлах устройств в диспетчере устройств

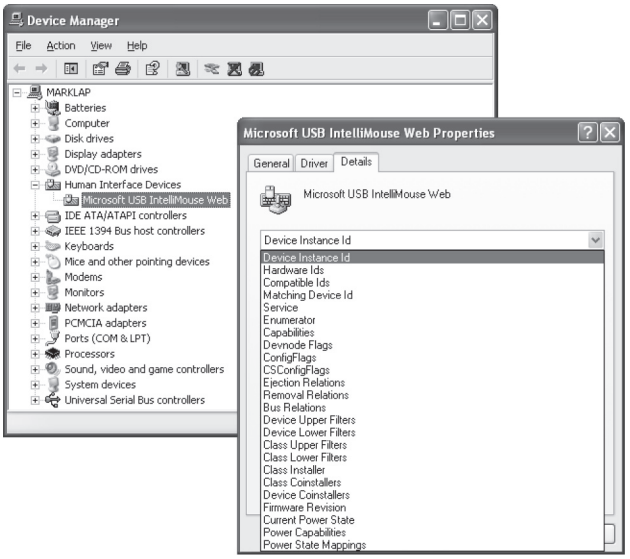
По умолчанию апплет Device Manager (Диспетчер устройств), доступный с вкладки Hardware (Оборудование) окна свойств системы, не показывает детальных сведений об узле устройств. Однако в Windows XP и Windows Server 2003 вы можете активизировать вкладку Details (Сведения), создав переменную окружения devmgr_show_details и присвоив ей значение 1. На этой вкладке отображается целый набор полей, в том числе идентификатор экземпляра устройства для узла, имя сервиса, фильтры и возможности в управлении электропитанием.

Самый простой способ запустить диспетчер устройств с вкладкой Details — открыть окно командной строки и ввести:

```
C:\>set devmgr_show_details=1
C:\>devmgmt.msc
```

см. след. стр.

На следующем экранном снимке показано, как выглядит содержимое этой вкладки для одного из устройств.



Параметр ClassGUID позволяет диспетчеру PnP найти раздел класса устройства в HKLM\SYSTEM\CurrentControlSet\Control\Class. Раздел класса клавиатур показан на рис. 9-29. Раздел, созданный для устройства в процессе перечисления, и раздел класса предоставляют диспетчеру PnP всю информацию, на основе которой он загружает драйверы, необходимые для узла данного устройства. Загрузка драйверов происходит в следующем порядке.

1. Любые низкоуровневые драйверы фильтров, указанные в параметре LowerFilters раздела, созданного для устройства в процессе перечисления.
2. Любые низкоуровневые драйверы фильтров, указанные в параметре LowerFilters раздела класса данного устройства.
3. Функциональный драйвер, заданный в параметре Service раздела, созданного для устройства в процессе перечисления. Значение этого параметра интерпретируется как имя раздела драйвера в HKLM\SYSTEM\CurrentControlSet\Services.
4. Любые высокоуровневые драйверы фильтров, указанные в параметре UpperFilters раздела, созданного для устройства в процессе перечисления.
5. Любые высокоуровневые драйверы фильтров, указанные в параметре UpperFilters раздела класса данного устройства.

Ссылки на драйверы всегда содержат имена их разделов в HKLM\SYSTEM\CurrentControlSet\Services.

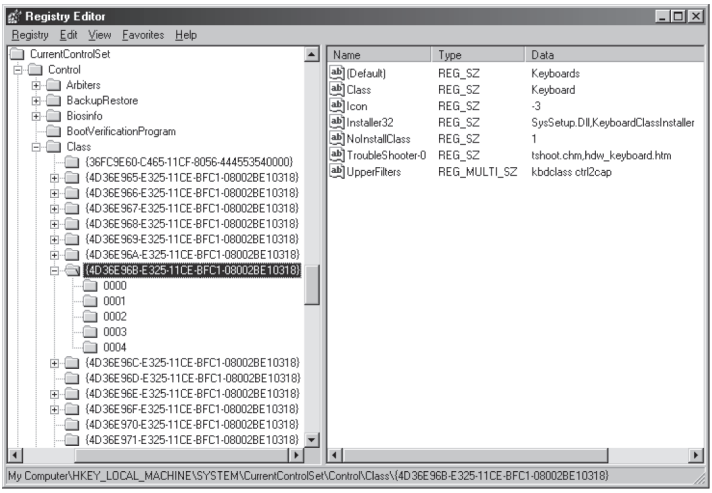


Рис. 9-29. *Раздел класса клавиатур*

ПРИМЕЧАНИЕ В DDK раздел, созданный для устройства в процессе перечисления, называется *аппаратным* (hardware key), а раздел класса — *программным* (software key).

Устройство «клавиатура», представленное на рис. 9-28 и 9-29, не имеет низкоуровневых драйверов фильтров. Его функциональный драйвер — i8042prt; в разделе класса клавиатур указано два высокоуровневых драйвера фильтров — kbdclass и ctrl2cap.

Установка драйвера

Если диспетчер PnP встречает устройство, драйвер которого не установлен, он обращается к диспетчеру PnP пользовательского режима, и тот устанавливает нужный драйвер. Если устройство обнаруживается при загрузке системы, для него определяется узел устройств, но загрузка драйверов откладывается до запуска диспетчера PnP пользовательского режима (он реализован в \Windows\System32\Umpnpmgr.dll и выполняется как сервис в процессе Services.exe).

Компоненты, участвующие в установке драйвера, показаны на рис. 9-30. Серые блоки на этом рисунке соответствуют компонентам, обычно предоставляемым системой, а остальные блоки — компонентам, предоставляемым установочными файлами. Сначала драйвер шины информирует диспетчер PnP о перечисленном устройстве, сообщая его DIID (1). Диспетчер PnP проверяет, определен ли в реестре подходящий функциональный драйвер. Если нет, он уведомляет диспетчер PnP пользовательского режима (2) о новом устройстве, сообщая его DIID. Диспетчер PnP пользовательского режима пытается автоматически установить драйверы для устройства. Если в процессе установки выводятся диалоговые окна, требующие внимания пользователя, а зарегистрированный в данный момент пользователь имеет привилегии

администратора (3), то диспетчер PnP пользовательского режима запускает Rundll32.exe (хост-программу апплетов Control Panel) для выполнения мастера установки оборудования (\Windows\System32\Newdev.dll). Если зарегистрированный в данный момент пользователь не имеет привилегий администратора (или в системе нет пользователей), а установка устройства требует взаимодействия с пользователем, диспетчер PnP пользовательского режима откладывает установку до того момента, когда в систему войдет привилегированный пользователь. Для поиска INF-файлов, соответствующих драйверам, совместимым с обнаруженным устройством, мастер установки оборудования использует API-функции Setup и CfgMgr (диспетчера конфигурации). При этом пользователю может быть предложено вставить в один из дисководов носитель с нужными INF-файлами; кроме того, просматриваются INF-файлы в \Windows\Driver Cache\i386\Driver.cab, где содержатся драйверы, поставляемые с Windows.

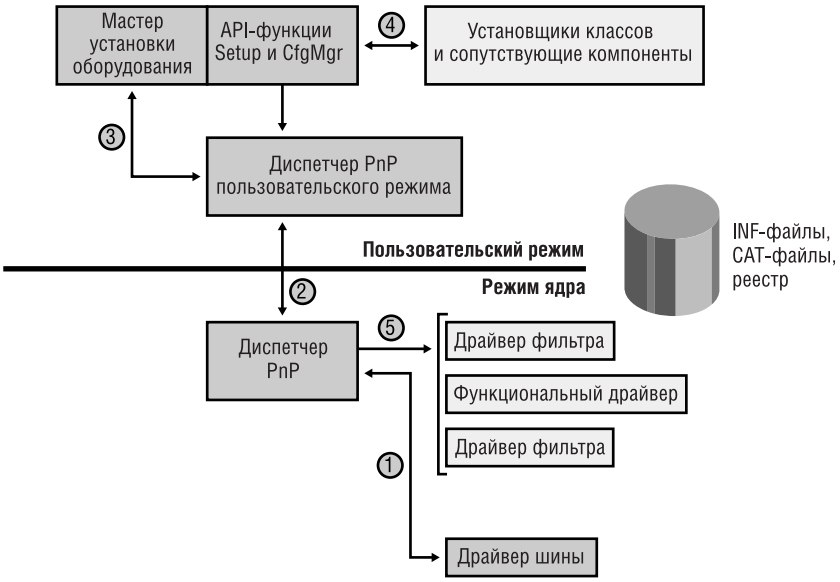


Рис. 9-30. *Компоненты, участвующие в установке драйвера*

Чтобы найти драйверы для нового устройства, процесс установки получает от драйвера шины список идентификаторов оборудования (hardware ID) и идентификаторов совместимых устройств (compatible ID). Эти идентификаторы описывают все способы, предусмотренные в установочном файле драйвера (INF-файле) для идентификации устройства. Списки упорядочиваются так, чтобы наиболее специфические характеристики устройства описывались первыми. Если совпадения идентификаторов обнаруживаются в нескольких INF-файлах, предпочтение отдается наиболее полным совпадениям. Аналогичным образом предпочтение отдается INF-файлам с цифровой подписью, а среди них — более новым. Если найденный идентификатор

соответствует идентификатору совместимого устройства, мастер установки оборудования может запросить носитель с обновленными драйверами для этого устройства.

INF-файл определяет местонахождение файлов функционального драйвера и содержит команды, которые вводят нужные данные в раздел перечисления и раздел класса драйвера. INF-файл может указать мастеру установки оборудования запустить DLL установщика класса или компонента, участвующего в установке устройства (4), — эти модули выполняют операции, специфичные для класса или устройства, например выводят диалоговые окна, позволяющие настраивать параметры устройства.

ЭКСПЕРИМЕНТ: просмотр INF-файла драйвера

При установке драйвера или другого программного обеспечения, у которого есть INF-файл, система копирует этот файл в каталог \Windows\Inf. Один из файлов, которые всегда будут в этом каталоге, — Keyboard.inf, поскольку это INF-файл для драйвера класса клавиатур. Просмотрите его содержимое, открыв в Notepad. Вы должны увидеть нечто вроде:

```
; Copyright (c) 1993-1996, Microsoft Corporation

[version]
signature="$Windows NT$"
Class=Keyboard
ClassGUID={4D36E96B-E325-11CE-BFC1-08002BE10318}
Provider=%MS%
LayoutFile=layout.inf
DriverVer=07/01/2001,5.1.2600.1106

[ClassInstall32.NT]
AddReg=keyboard_class_addreg
...
```

Если вы проведете поиск в этом файле по «.sys», то обнаружите запись, указывающую диспетчеру PnP пользовательского режима установить драйверы i8042prt.sys и kbdclass.sys:

```
...
[STANDARD_CopyFiles]
i8042prt.sys
kbdclass.sys
...
```

Перед установкой драйвера диспетчер PnP пользовательского режима проверяет системную политику проверки цифровых подписей в драйверах. Эта политика хранится в разделе реестра HKLM\SOFTWARE\Microsoft\Driver Signing\Policy, если администратор выбрал общесистемную политику, или в HKCU\Software\Microsoft\Driver Signing\Policy, если в системе применяются

политики только по отношению к индивидуальным пользователям. Политика проверки цифровых подписей в драйверах настраивается через диалоговое окно Driver Signing Options (Параметры подписывания драйвера), доступное с вкладки Hardware (Оборудование) окна свойств системы (рис. 9-31). Если указанные параметры заставляют систему блокировать установку неподписанных драйверов или предупреждать о таких попытках, диспетчер PnP пользовательского режима проверяет в INF-файле драйвера запись, указывающую на каталог (файл с расширением .cat), где содержится цифровая подпись драйвера.

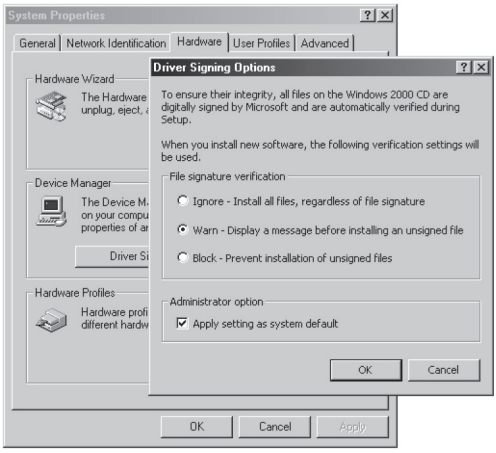


Рис. 9-31. *Параметры проверки цифровых подписей в драйверах*

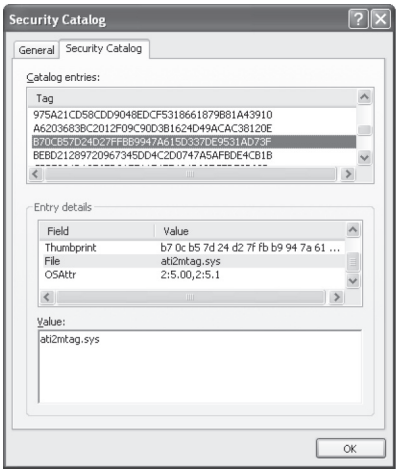
Лаборатория Microsoft WHQL тестирует драйверы, поставляемые с Windows и предлагаемые изготовителями оборудования. Драйвер, прошедший тесты WHQL, «подписывается» Microsoft. Это означает, что создается *хэш*, или уникальное значение, представляющее файлы драйвера, в том числе его образ, а затем этот хэш подписывается с применением закрытого ключа Microsoft, предназначенного для подписания драйверов. Подписанный хэш помещается в CAT-файл и записывается на дистрибутив Windows или передается изготовителю, который включает его в свой драйвер.

ЭКСПЕРИМЕНТ: просмотр CAT-файлов

При установке компонента, например драйвера, файлы которого включают CAT-файл, Windows копирует этот файл в подкаталог каталога \Windows\System32\Catroot. Перейдите в этот каталог с помощью Explorer и найдите подкаталог с CAT-файлами. В частности, в Nt5.cat и Nt5inf.cat хранятся подписи для системных файлов Windows.

Открыв один из CAT-файлов, вы увидите диалоговое окно с двумя вкладками: General (Общие), на которой показывается информация о подписи в данном файле, и Security Catalog (Каталог безопасности), где представлены хэши компонентов, подписанных с использованием

этого CAT-файла. Ниже дан пример CAT-файла для видеодрайверов ATI, где приведено содержимое хэша для минипорт-драйверов видеоадаптера. Остальные хэши в этом файле относятся к вспомогательным DLL, поставляемым с данными драйверами.



При установке драйвера диспетчер PnP пользовательского режима извлекает из CAT-файла подпись драйвера, расшифровывает ее с применением открытого ключа Microsoft и сравнивает полученный в результате хэш с хэшем файла устанавливаемого драйвера. Если хэши совпадают, драйвер считается проверенным на соответствие требованиям WHQL. Если проверка заканчивается неудачно, диспетчер PnP пользовательского режима действует так, как это диктует действующая политика: запрещает установку, предупреждает пользователя о том, что драйвер не подписан, или автоматически устанавливает драйвер.

ПРИМЕЧАНИЕ Наличие подписей у драйверов, устанавливаемых программами установки, которые самостоятельно настраивают реестр и копируют файлы драйвера в систему, а также у драйверов, динамически загружаемых приложениями, не проверяется. Политика проверки подписей драйверов распространяется только на драйверы, устанавливаемые с помощью INF-файлов.

После установки драйвера диспетчер PnP режима ядра (этап 5 на рис. 9-30) запускает драйвер и вызывает его процедуру добавления устройства, чтобы уведомить драйвер о присутствии устройства, для управления которым он был загружен. Далее формируется узел устройства, как мы уже объясняли.

ПРИМЕЧАНИЕ В Windows XP и Windows Server 2003 диспетчер PnP пользовательского режима также проверяет, не включен ли устанавливаемый драйвер в *защищенный список драйверов* (protected driver list), поддерживаемых Windows Update, и, если включен, блокирует установ-

ку с выводом предупреждения для пользователя. В этот список вносятся драйверы, которые имеют известные ошибки или просто несовместимы, и их установка блокируется. Детали см. по ссылке *www.microsoft.com/whdc/winlogo/drvsign/drv_protect.mspx*.

Диспетчер электропитания

Как и PnP-функции Windows, управление электропитанием требует аппаратной поддержки. Она должна отвечать спецификации Advanced Configuration and Power Interface (ACPI) (см. *www.teleport.com/~acpi/spec.htm*). Согласно этой спецификации BIOS (Basic Input Output System) тоже должна соответствовать стандарту ACPI. Этим требованиям удовлетворяет большинство x86-компьютеров, выпускавшихся с конца 1998 года.

ПРИМЕЧАНИЕ Некоторые компьютеры — особенно те, которые были изготовлены несколько лет назад, — не полностью совместимы со стандартом ACPI. Они соответствуют более старому стандарту Advanced Power Management (APM), определяющему меньшее количество функций управления электропитанием, чем ACPI. Windows поддерживает ограниченный набор функций управления электропитанием для APM-систем, но мы не будем вдаваться в детали этого стандарта и основное внимание уделим поведению Windows, установленной на ACPI-совместимых компьютерах.

Стандарт ACPI определяет различные уровни энергопотребления для системы и устройств. Шесть состояний для системы — от S0 (полностью активное, или рабочее, состояние) до S5 (полное отключение) — перечислены в таблице 9-3. Каждое из них характеризуется следующими параметрами.

- **Энергопотребление (power consumption)** Количество энергии, потребляемой компьютером.
- **Возобновление работы ПО (software resumption)** Состояние программного обеспечения при переходе компьютера в «более активное» состояние.
- **Аппаратная задержка (hardware latency)** Время, необходимое на то, чтобы вернуть компьютер в полностью активное состояние.

Таблица 9-3. *Определения состояний системы с различным энергопотреблением*

Состояние	Энергопотребление	Возобновление работы ПО	Аппаратная задержка
S0 (fully on) (полностью активное состояние)	Максимальное	—	Нет
S1 (sleeping) (состояние ожидания)	Менее S0, но более S2	Система возобновляет работу с той точки, где она была прервана (возвращается в состояние S0)	Менее 2 секунд

Таблица 9-3. (*окончание*)

Состояние	Энергопотребление	Возобновление работы ПО	Аппаратная задержка
S2 (sleeping) (состояние ожидания)	Менее S1, но более S3	Система возобновляет работу с той точки, где она была прервана (возвращается в состояние S0)	От 2 секунд и более
S3 (sleeping) (состояние ожидания)	Менее S2, процессор отключен	Система возобновляет работу с той точки, где она была прервана (возвращается в состояние S0)	От 2 секунд и более
S4 (hibernating) (спящий режим)	Ток подается на кнопку включения электропитания и в контур пробуждения (wake circuitry)	Система перезапускается с помощью файла спящего режима (hibernate file) и возобновляет работу с той точки, где она была прервана переходом в спящий режим (возвращается в состояние S0)	Длительная, неопределенная
S5 (fully off) (полное отключение)	Ток подается на кнопку включения электропитания	Система загружается заново	Длительная, неопределенная

В состояния ожидания (S1–S4) компьютер кажется отключенным, так как потребляет меньше энергии. Но при этом он сохраняет и в памяти, и на диске всю информацию, необходимую для возврата в состояние S0. В состояниях S1–S3 для сохранения содержимого памяти нужно достаточное количество энергии, поскольку при переходе в S0 (при пробуждении компьютера пользователем или устройством) диспетчер электропитания возобновляет работу системы с той точки, где оно было прервано. Когда система переходит в состояние S4, диспетчер электропитания сохраняет содержимое памяти в сжатой форме в файле спящего режима (Hiberfil.sys), который помещается в корневой каталог системного тома. (Этот файл должен быть такого размера, чтобы в нем могло уместиться несжатое содержимое всей памяти; сжатие используется для того, чтобы свести к минимуму операции ввода-вывода на диске, а также ускорить переход в спящий режим и выход из него.) Сохранив содержимое памяти, диспетчер электропитания отключает компьютер. При последующем включении компьютера происходит обычный процесс загрузки — с тем исключением, что Ntldr проверяет наличие действительного образа памяти, сохраненного в файле спящего режима. Если в этом файле сохранены данные о состоянии системы, Ntldr считывает его содержимое в память и возобновляет выполнение с точки, зафиксированной в Hiberfil.sys.

Компьютер никогда не переходит напрямую между состояниями S1 и S4, для этого ему нужно сначала перейти в состояние S0. Как показано на рис. 9-32, переход системы из состояний S1–S5 в состояние S0, называется

пробуждением (waking), а переход из состояния S0 в состояния S1–S5 — *переходом в сон* (sleeping).

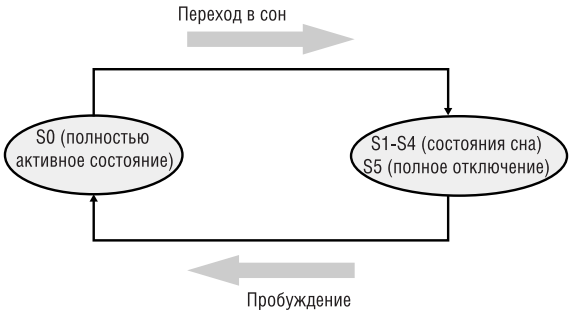


Рис. 9-32. *Переходы системы между различными состояниями энергопотребления*

Хотя система может пребывать в одном из шести состояний энергопотребления, ACPI определяет для устройств четыре состояния: D0–D3. В состоянии D0 устройство полностью включено, а в состоянии D3 полностью отключено. ACPI позволяет драйверам и устройствам самостоятельно определять состояния D1 и D2 с единственным условием, что устройство в состоянии D1 должно потреблять столько же или меньше энергии, чем в состоянии D0, а в состоянии D2 — столько же или меньше, чем в состоянии D1. Microsoft совместно с крупными OEM-производителями определила набор спецификаций управления электропитанием (см. *www.microsoft.com/whdc/resources/respec/specs/pmref*), в которых описываются состояния энергопотребления для всех устройств конкретного класса (основные классы устройств: видеоадаптеры, сеть, SCSI и т. д.). Некоторые устройства могут быть лишь включены или выключены, поэтому для них промежуточные состояния не определены.

Работа диспетчера электропитания

Политика управления электропитанием в Windows определяется диспетчером электропитания и драйверами устройств. Владельцем системной политики управления электропитанием является диспетчер электропитания. Это значит, что он принимает решение о том, в каком состоянии энергопотребления должна находиться система в текущий момент. При необходимости выключения либо перехода в ждущий или спящий режим диспетчер электропитания указывает устройствам, поддерживающим управление электропитанием, перейти в соответствующее состояние. Этот диспетчер принимает решение о переходе в другое состояние энергопотребления, исходя из:

- уровня активности системы;
- уровня заряда аккумуляторов;
- наличия запросов приложений на выключение компьютера или переход в ждущий/спящий режим;

- действий пользователя, например нажатия кнопки включения электропитания;
- параметров электропитания, заданных в Control Panel.

Часть информации, получаемой диспетчером PnP при перечислении устройств, связана с поддержкой устройствами функций управления электропитанием. Драйвер сообщает, поддерживает ли устройство состояния D1 и D2, а также какие задержки требуются ему для перехода из состояний D1–D3 в D0 (последняя часть данных необязательна). Чтобы диспетчеру было легче определять, когда систему следует переводить в другое состояние энергопотребления, драйверы шин также возвращают таблицу сопоставлений между системными состояниями (S0–S5) и состояниями, поддерживаемыми конкретным устройством. В этой таблице указывается состояние устройства с наименьшим энергопотреблением для каждого системного состояния. В таблице 9-4 показан пример таблицы сопоставлений для шины, поддерживающей все четыре возможных состояния устройств. Большинство драйверов полностью выключают свои устройства (D3) при выходе системы из состояния S0, чтобы свести к минимуму энергопотребление, пока машина не используется. Однако некоторые устройства вроде сетевых адаптеров поддерживают функцию вывода системы из состояний сна. О наличии подобной функции также сообщается при перечислении устройств.

Таблица 9-4. *Пример таблицы сопоставлений системных состояний с состояниями устройств*

Состояние системы	Состояние устройства
S0 (полностью активное)	D0 (полностью активное)
S1 (ждущий режим)	D2
S2 (ждущий режим)	D2
S3 (ждущий режим)	D2
S4 (спящий режим)	D3 (полное отключение)
S5 (полное отключение)	D3 (полное отключение)

Участие драйверов в управлении электропитанием

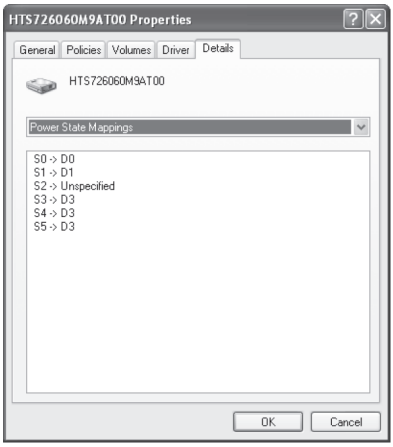
Диспетчер электропитания, принимая решение о переходе системы в другое состояние, посылает команды процедуре драйвера, отвечающей за диспетчеризацию электропитания. Управлять устройством могут несколько драйверов, но только один из них является владельцем политики управления электропитанием устройства. Этот драйвер определяет состояние устройства в зависимости от состояния энергопотребления системы. Например, при переходе системы из состояния S0 в состояние S1 драйвер может принять решение о переводе устройства из состояния D0 в состояние D1. Вместо того чтобы напрямую оповещать об этом другие драйверы, участвующие в управлении устройством, владелец политики управления электропитанием устройства делает это через диспетчер электропитания, вызывая функцию *PoRequestPowerIrp*.

Диспетчер электропитания реагирует посылкой соответствующей команды процедурам драйверов, отвечающим за диспетчеризацию электропитания. Такое поведение позволяет диспетчеру контролировать число активных команд управления электропитанием в системе: включение некоторых устройств требует значительного количества электроэнергии, поэтому активизация сразу нескольких таких устройств недопустима.

ЭКСПЕРИМЕНТ: просмотр сопоставлений состояний электропитания в драйвере

В Windows XP и Windows Server 2003 это можно сделать с помощью диспетчера устройств. Откройте окно свойств для какого-либо устройства и выберите запись Power State Mappings (Сопоставления энергосбережения) в раскрывающемся списке на вкладке Details (Сведения). (По умолчанию диспетчер устройств не показывает эту вкладку. Как ее включить, см. в эксперименте «просмотр детальных сведений об узлах устройств в диспетчере устройств» ранее в этой главе.)

На иллюстрации ниже показаны такие сопоставления для драйвера диска. Кроме состояний D0 (полное включение) и D3 (полное отключение), он поддерживает промежуточное состояние D1, сопоставленное с S1.



Для многих команд управления электропитанием предусмотрены соответствующие команды-запросы. Так, при переходе системы в ждущий режим, диспетчер электропитания сначала опрашивает устройства о допустимости такого перехода. Устройство, занятое выполнением критичных по времени операций или взаимодействующее с другим аппаратным устройством, может отклонить запрос, и система останется в прежнем состоянии.

ЭКСПЕРИМЕНТ: просмотр возможностей и системной политики управления электропитанием

Вы можете выяснить возможности своего компьютера в управлении электропитанием с помощью команды *!pocaps* отладчика ядра. Ниже приведен пример вывода этой команды для ACPI-совместимого портативного компьютера с Windows Professional.

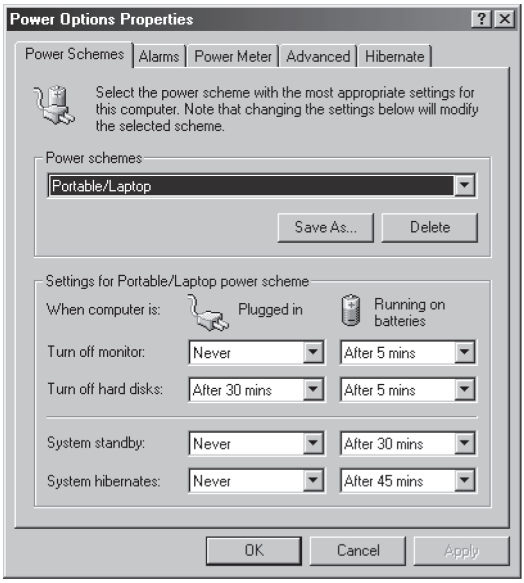
```
kd> !pocaps
PopCapabilities @ 0x8046adc0
  Misc Supported Features:  PwrButton SlpButton Lid S1 S3 S4 S5
                            HiberFile FullWake
  Processor Features:       Thermal Throttle (MinThrottle = 03,
                            Scale = 08)
  Disk Features:            SpinDown
  Battery Features:         BatteriesPresent
    Battery 0 - Capacity: 00000000  Granularity: 00000000
    Battery 1 - Capacity: 00000000  Granularity: 00000000
    Battery 2 - Capacity: 00000000  Granularity: 00000000
  Wake Caps
    Ac OnLine Wake:         Sx
    Soft Lid Wake:          Sx
    RTC Wake:               S3
    Min Device Wake:        Sx
    Default Wake:           Sx
```

Строка Misc Supported Features сообщает, что кроме S0 данная система поддерживает состояния S1, S3, S4 и S5 (S2 не реализовано) и имеет действительный файл спящего режима, в который можно сохранить содержимое системной памяти при переходе в спящий режим (состояние S4).

Диалоговое окно Power Options Properties (Свойства: Электропитание), показанное на следующей иллюстрации (оно открывается через Control Panel), позволяет настроить различные аспекты системной политики управления электропитанием. Конкретные параметры, доступные для настройки, зависят от степени поддержки системой функций управления электропитанием.

ACPI-совместимый портативный компьютер с Windows Professional или Home предоставляет максимум возможностей в управлении электропитанием. В таких системах можно установить интервалы простоя, по истечении которых отключается монитор, останавливаются жесткие диски и осуществляется переход в ждущий (состояние S1) и спящий режимы (состояние S4). Кроме того, вкладка Advanced (Дополнительно) в диалоговом окне Power Options Properties позволяет указать поведение системы при нажатии кнопок включения электропитания и перехода в спящий режим, а также при закрытии крышки ноутбука.

см. след. стр.



Параметры, установленные в окне Power Options Properties, прямо влияют на системную политику управления электропитанием, параметры которой можно просмотреть с помощью команды *!popolicy* отладчика ядра. Вот как выглядит информация, сообщаемая этой командой для той же системы:

```
kd> !popolicy
SYSTEM_POWER_POLICY (R.1) @ 0x80469180
  PowerButton:         Off  Flags: 00000003   Event: 00000000   Query UI
  SleepButton:       Sleep  Flags: 00000003   Event: 00000000   Query UI
  LidClose:      Hibernate  Flags: 00000001   Event: 00000000   Query
  Idle:               None  Flags: 00000001   Event: 00000000   Query
  OverThrottled:     Sleep  Flags: c0000004   Event: 00000000   Override
                     NoWakes Critical
  IdleTimeout:    00000000  IdleSensitivity:        32
  MinSleep:             S1  MaxSleep:               S3
  LidOpenWake:          S0  FastSleep:              S1
  WinLogonFlags:  00000000  S4Timeout:        00000000
  VideoTimeout:   00000000  VideoDim:               6e
  SpinTimeout:    00000708  OptForPower:            01
  FanTolerance:         64  ForcedThrottle:         64
  MinThrottle:          19
```

Первые строки описывают, как будет реагировать система на нажатие кнопок включения электропитания и перехода в спящий режим. В данной системе нажатие кнопки включения электропитания интерпретируется как выключение электропитания, нажатие кнопки перехода в спящий режим переводит систему в ждущий режим, а закрытие крышки ноутбука вызывает переход в спящий режим.

Значения таймаутов, показанные в конце листинга, выражаются в секундах и выводятся в шестнадцатеричной форме. Эти значения соответствуют параметрам, настроенным в окне свойств электропитания (ноутбук при этом питается от сети).

Как драйвер управляет электропитанием устройства

Драйвер не только отвечает на команды диспетчера электропитания, связанные с изменением состояния системы, но и может сам управлять состоянием энергопотребления своих устройств. В некоторых случаях драйвер может снизить энергопотребление управляемого им устройства, если оно неактивно в течение определенного времени. Драйвер может обнаруживать простаивающие устройства самостоятельно или через механизмы, предоставляемые диспетчером электропитания. Во втором случае устройство регистрируется в диспетчере электропитания вызовом функции *PoRegisterDeviceForIdleDetection*. Эта функция сообщает диспетчеру электропитания пороговые интервалы простоя устройства и указывает, в какое состояние следует переводить устройство, если оно простаивает. Драйвер задает два таймаута: первый — для энергосберегающей конфигурации, второй — для максимально производительной. Вызвав *PoRegisterDeviceForIdleDetection*, драйвер должен уведомлять диспетчер электропитания об активности устройства через функцию *PoSetDeviceBusy*.

Резюме

Подсистема ввода-вывода определяет модель обработки ввода-вывода в Windows и предоставляет функции, необходимые многим драйверам. Основная сфера ее ответственности — создание IRP, представляющих запросы ввода-вывода, передача этих пакетов через различные драйверы и возврат результатов вызывающему потоку по завершении ввода-вывода. Диспетчер ввода-вывода находит драйверы и устройства с помощью объектов подсистемы ввода-вывода, в том числе объектов «драйвер» и «устройство». Для большего быстродействия подсистема ввода-вывода Windows всегда работает асинхронно — даже при обработке синхронного ввода-вывода, запрошенного из пользовательского режима.

К драйверам устройств относятся не только традиционные драйверы, управляющие аппаратными устройствами, но и драйверы файловой системы, сетевые драйверы, а также многоуровневые драйверы фильтров. Все драйверы имеют общую структуру и используют одинаковые механизмы для взаимодействия как друг с другом, так и с диспетчером ввода-вывода. Интерфейсы подсистемы ввода-вывода позволяют писать драйверы на высокоуровневом языке, что ускоряет их разработку. Поскольку драйверы имеют общую структуру, они могут располагаться один над другим, а это обеспечивает модульность и уменьшает дублирование функций между драйверами. Все драй-

веры устройств, создаваемые для Windows, должны разрабатываться с учетом необходимости корректной работы в многопроцессорных системах.

Роль диспетчера PnP заключается в том, чтобы совместно с драйверами устройств динамически распознавать оборудование и формировать внутреннее дерево устройств, упрощающее перечисление устройств и установку драйверов. Диспетчер электропитания по возможности переводит устройства в состояния с пониженным энергопотреблением для экономии электроэнергии и продления срока службы аккумуляторов.

Следующие четыре главы мы посвятим смежной тематике: управлению устройствами внешней памяти, файловым системам (особое внимание будет уделено NTFS), диспетчеру кэша и поддержке сетей.

ГЛАВА 10

Управление внешней памятью

Термин *внешняя память* (storage) относится к носителям, применяемым в самых разнообразных устройствах, в том числе к магнитным лентам, оптическим дискам, гибким дискам, локальным жестким дискам и сети устройств хранения данных (storage area networks, SAN). Windows предоставляет специализированную поддержку для каждого класса носителей внешней памяти. Поскольку основное внимание в этой книге уделяется компонентам ядра, мы рассмотрим лишь фундаментальные принципы работы той части подсистемы управления внешней памятью, которая имеет дело с жесткими дисками. Существенную часть поддержки сменных носителей и удаленных устройств внешней памяти Windows реализует в пользовательском режиме.

В этой главе мы исследуем, как драйверы устройств режима ядра взаимодействуют с драйверами файловой системы и дисками. Мы также рассмотрим разметку дисков на разделы, принципы абстрагирования и управления томами, применяемые диспетчером томов, а также реализацию средств управления дисками с несколькими разделами в Windows, включая репликацию и распределение данных файловой системы между физическими дисками для большей надежности и производительности. В заключение мы опишем, как драйверы файловой системы монтируют свои тома.

Базовая терминология

Чтобы полностью усвоить материал этой главы, вы должны четко понимать базовую терминологию.

- *Диск* — физическое устройство внешней памяти, например жесткий диск, 3,5-дюймовая дискета или компакт-диск (CD-ROM).
- Диск делится на *секторы*, блоки фиксированного размера. Размер сектора определяется аппаратно. Например, размер сектора жесткого диска, как правило, составляет 512 байтов, а размер сектора CD-ROM — обычно 2048 байт.
- *Раздел* (partition) — набор непрерывных секторов на диске. Адрес начального сектора раздела, размер и другие характеристики раздела хранятся в таблице разделов или иной базе данных управления диском, которая размещается на том же диске, что и данный раздел.

- *Простой том* (simple volume) — объект, представляющий секторы одного раздела, которым драйверы файловых систем управляют как единым целым.
- *Составной том* (multipartition volume) — объект, представляющий секторы нескольких разделов, которыми драйверы файловых систем управляют как единым целым. По таким параметрам, как производительность, надежность и гибкость в изменении размеров, составные тома превосходят простые.

Драйверы дисков

Драйверы устройств, участвующие в управлении конкретным устройством внешней памяти (накопителем), обобщенно называются стеком драйверов внешней памяти (storage stack). На рис. 10-1 показаны все типы драйверов, которые могут присутствовать в стеке. В этой главе мы описываем поведение драйверов устройств, расположенных в стеке ниже уровня файловой системы. (О драйвере файловой системы см. главу 12.)

Рис. 10-1. *Стек драйверов устройств внешней памяти в Windows*

Ntldr

Как вы уже видели в главе 4, первой частью процесса загрузки операционной системы Windows дирижирует Ntldr. Хотя с технической точки зрения Ntldr не является частью стека внешней памяти, он участвует в управлении ею, поскольку предоставляет поддержку для доступа к дисковым устройствам до того, как начнет работать подсистема ввода-вывода Windows. Он находится на системном томе и запускается кодом, размещенным в загрузочном секторе этого тома. Ntldr считывает с системного тома файл Boot.ini и предлагает пользователю выбрать вариант загрузки. Имена разделов в Boot.ini представлены в виде multi(0)disk(0)rdisk(0)partition(1). Эти имена являются частью стандартной схемы именования разделов Advanced RISC Computing (ARC), используемой микрокодом Alpha и других RISC-процессоров. Ntldr транслирует имя выбранного пользователем элемента Boot.ini в имя загрузочного раздела и загружает в память системные файлы Windows (начиная с реестра, Ntoskrnl.exe и загрузочных драйверов). Во всех случаях Ntldr использует BIOS для чтения диска, содержащего системный том, но, как описано в главе 4, иногда полагается на функции минипорт-драйвера диска для чтения с диска, где находится загрузочный том.

Драйвер класса дисков, порт- и минипорт-драйверы

При инициализации диспетчер ввода-вывода запускает драйверы жестких дисков. Драйверы устройств внешней памяти в Windows соответствуют архитектуре «класс-порт-минипорт». Согласно этой архитектуре, Microsoft предоставляет драйвер класса внешней памяти, который реализует функциональность, общую для всех устройств внешней памяти, и порт-драйвер, который поддерживает функциональность, общую для конкретной шины, например SCSI (Small Computer System Interface) или IDE (Integrated Device Electronics). А изготовители оборудования поставляют минипорт-драйверы, подключаемые к порт-драйверам и формирующие интерфейс между Windows и конкретными устройствами.

В архитектуре драйверов дисковой памяти только драйверы класса имеют стандартные интерфейсы драйверов устройств Windows. Минипорт-драйверы вместо интерфейса драйверов устройств используют интерфейс порт-драйверов, который просто реализует набор процедур, служащих интерфейсом между Windows и минипорт-драйверами. Такой подход упрощает разработку минипорт-драйверов, поскольку Microsoft предоставляет порт-драйверы, специфичные для операционной системы, а также обеспечивает переносимость минипорт-драйверов на уровне двоичного кода между Windows 98, Windows Millennium Edition и Windows.

Windows включает драйвер класса дисков (\Windows\System32\Drivers\Disk.sys), реализующий стандартную функциональность дисков. Windows также предоставляет разнообразные порт-драйверы дисков. Например, Scsiport.sys — это порт-драйвер дисков, подключаемых к SCSI-шине, а Atapi.sys — порт-драйвер для систем на базе IDE. В Windows Server 2003 введен порт-

драйвер Storport.sys, заменяющий Scsiport.sys. Storport.sys был разработан для реализации функциональности высокопроизводительных аппаратных RAID-контроллеров и адаптеров Fibre Channel. Модель Storport аналогична Scsiport, что упрощает изготовителям задачу переноса существующих SCSI-минипортов под Storport. Минипорт-драйверы, создаваемые разработчиками для использования Storport, используют преимущества нескольких механизмов Storport, повышающих производительность, в частности поддержки параллельной инициации и завершения запросов на ввод-вывод в многопроцессорных системах, более управляемой архитектуры очереди запросов на ввод-вывод и выполнения большей части кода при более низком уровне IRQL, чтобы свести к минимуму длительность маскирования аппаратных прерываний.

Драйверы Scsiport.sys и Atapi.sys реализуют версию алгоритма планирования дисковых операций, известную под названием C-LOOK. Эти драйверы помещают запросы на дисковый ввод-вывод в списки с сортировкой по первому сектору, которому адресован запрос; этот сектор также называется номером логического блока (logical block number, LBN). С помощью функций *KeInsertByKeyDeviceQueue* и *KeRemoveByKeyDeviceQueue* (документированных в Windows DDK) они представляют запросы ввода-вывода как элементы (items) и используют начальный сектор запроса в качестве ключа, требуемого этими функциями. Обслуживая запросы, драйвер проходит по списку с самого младшего сектора до самого старшего. Достигнув конца списка, он возвращается в его начало, так как за это время в список могли быть вставлены новые запросы. Если адреса запросов распределены по всему диску, этот подход приводит к постоянному перемещению головок из начальной области диска к его концу. Storport.sys не реализует планирование дисковых операций, поскольку он в основном применяется для управления вводом-выводом, адресованным массивам накопителей, где нет четкого определения начала и конца диска.

С Windows поставляются некоторые минипорт-драйверы, включая Aha154x.sys для SCSI-контроллеров семейства Adaptec 1540. В системах, где установлено минимум одно IDE-устройство на основе ATAPI, функциональность минипортов предоставляют драйверы Pciidex.sys и Pciide.sys. Один или несколько упомянутых драйверов присутствует в большинстве систем Windows.

Драйверы iSCSI

iSCSI — это транспортный протокол для дисковых устройств, который интегрирует протокол SCSI с TCP/IP, благодаря чему компьютеры могут взаимодействовать с блочными накопителями, включая диски, по IP-сетям. Архитектура сети устройств хранения данных (storage area networking, SAN) обычно базируется на сети Fibre Channel, но администраторы могут использовать iSCSI для создания сравнительно недорогих SAN на основе таких сетевых технологий, как гигабитная Ethernet, что позволяет обеспечить масштабируемость, защиту от катастроф, эффективное резервное копирование и защиту данных. В Windows поддержка iSCSI реализуется в виде Microsoft

iSCSI Software Initiator, который можно скачать с сайта Microsoft и который работает в Windows 2000, Windows XP и Windows Server 2003.

Microsoft iSCSI Software Initiator включает несколько компонентов.

- **Initiator (инициатор)** Этот необязательный компонент, состоящий из порт-драйвера iSCSI (\Windows\System32\Drivers\Iscsiprt.sys) и минипорт-драйвера (\Windows\System32\Drivers\Msiscis.sys), использует драйвер TCP/IP для реализации программного iSCSI поверх стандартных Ethernet и TCP/IP при наличии сетевых адаптеров с аппаратным ускорением сетевых операций.
- **Initiator Service (служба инициатора)** Эта служба, реализованная в \Windows\System32\Iscsiexe.exe, управляет обнаружением и защитой всех инициаторов iSCSI, а также инициацией и завершением сеансов. Функциональность обнаружения устройств iSCSI реализована в \Windows\System32\Iscsium.dll и соответствует спецификации протокола Internet Storage Name Service (iSNS).
- **Управляющие приложения** К ним относятся Iscsicli.exe (утилита командной строки для управления соединениями iSCSI-устройств и их защитой) и соответствующий апплет для Control Panel (Панель управления).

Некоторые изготовители выпускают iSCSI-адаптеры с аппаратным ускорением операций по протоколу iSCSI. Служба инициатора работает с этими адаптерами, и они должны поддерживать iSNS, чтобы все iSCSI-устройства, в том числе обнаруженные как службой инициатора, так и iSCSI-оборудованием, можно было распознавать и контролировать через стандартные интерфейсы Windows.

MPIO-драйверы

У большинства дисковых устройств только один *путь* (path) между ними и компьютером — набор адаптеров, кабелей и коммутаторов. В серверах, требующих высокого уровня готовности к работе, применяются решения с несколькими путями — между компьютером и диском существует более одного набора соединительного оборудования, чтобы при аварии одного пути система все равно могла бы обращаться к диску по альтернативному пути. Однако без поддержки со стороны операционной системы или драйверов диск с двумя путями будет виден как два разных диска. Windows включает поддержку ввода-вывода по нескольким путям (multipath I/O, MPIO) для управления дисками с несколькими путями как одним диском; эта поддержка опирается на сторонние драйверы — модули, специфичные для конкретного устройства (device-specific modules, DSM). Эти модули берут на себя всю специфику управления путями, в том числе политику балансировки нагрузки, на основе которой выбирается путь передачи запросов ввода-вывода, и механизмы обнаружения ошибок, уведомляющие Windows об аварии того или иного пути. Поддержка MPIO доступна для Windows 2000 Server, Advanced Server, Datacenter Server и Windows Server 2003 в виде Microsoft MPIO

Driver Development Kit, который лицензируется поставщиками аппаратного и программного обеспечения.

В стеке драйверов внешней памяти Windows MPIO (рис. 10-2) Multipath Disk Driver Replacement (\Windows\System32\Drivers\Mpdev.sys) заменяет функциональность стандартного драйвера класса Disk.sys. Mpdev.sys захватывает во владение объект «устройство», представляющий диски с несколькими путями, чтобы для таких дисков существовал лишь один объект «устройство». Кроме того, этот драйвер отвечает за поиск подходящего DSM для управления путями к устройству. Multipath Bus Driver (\Windows\System32 \Drivers \ Mpio.sys) управляет соединениями между компьютером и устройством, в том числе обеспечивая управление электропитанием данного устройства. Mpdev.sys уведомляет Mpio.sys о наличии устройств, которые тот должен контролировать. Наконец, Multipath Port Filter (\Windows\System32 \Drivers\Mpsfltr.sys) размещается поверх порт-драйвера для диска с несколькими путями и управляет информацией, передаваемой вверх по стеку устройств.

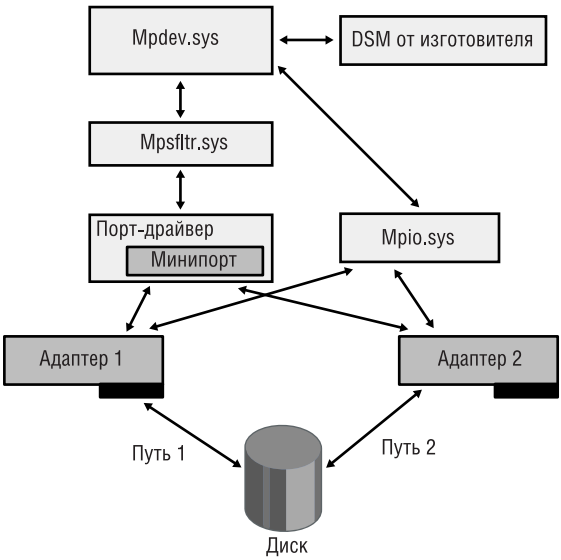
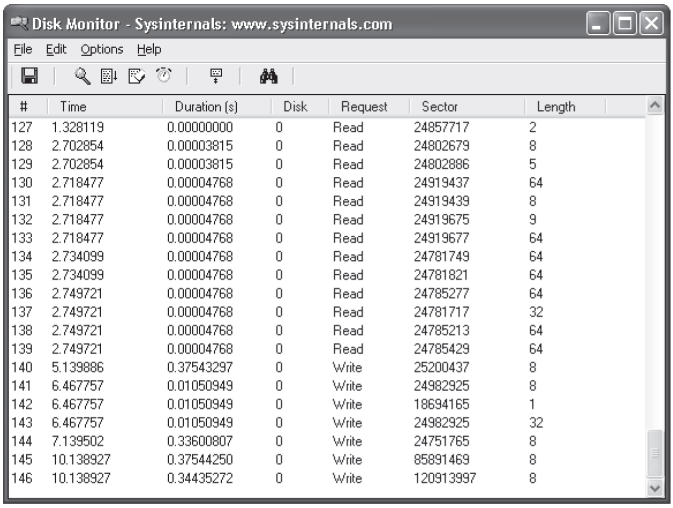


Рис. 10-2. *Стек устройств внешней памяти Windows MPIO*

ЭКСПЕРИМЕНТ: наблюдение за вводом-выводом на физическом диске

С помощью механизма Event Tracing for Windows (см. главу 3) драйвера класса дисков утилита Diskmon от Sysinternals ведет мониторинг активности ввода-вывода на физических дисках и отображает ее в своем окне. Содержимое этого окна обновляется раз в секунду. Для каждой операции Diskmon показывает время, длительность, номер целевого диска, тип и смещение, а также длину.



#	Time	Duration (s)	Disk	Request	Sector	Length
127	1.328119	0.00000000	0	Read	24857717	2
128	2.702854	0.00003815	0	Read	24802679	8
129	2.702854	0.00003815	0	Read	24802886	5
130	2.718477	0.00004768	0	Read	24919437	64
131	2.718477	0.00004768	0	Read	24919439	8
132	2.718477	0.00004768	0	Read	24919675	9
133	2.718477	0.00004768	0	Read	24919677	64
134	2.734099	0.00004768	0	Read	24781749	64
135	2.734099	0.00004768	0	Read	24781821	64
136	2.749721	0.00004768	0	Read	24785277	64
137	2.749721	0.00004768	0	Read	24781717	32
138	2.749721	0.00004768	0	Read	24785213	64
139	2.749721	0.00004768	0	Read	24785429	64
140	5.139886	0.37543297	0	Write	25200437	8
141	6.467757	0.01050949	0	Write	24982925	8
142	6.467757	0.01050949	0	Write	18694165	1
143	6.467757	0.01050949	0	Write	24982925	32
144	7.139502	0.33600807	0	Write	24751765	8
145	10.138927	0.37544250	0	Write	85891469	8
146	10.138927	0.34435272	0	Write	120913997	8

Объекты «устройство» для дисков

Драйвер класса дисков создает объекты «устройство», представляющие диски и дисковые разделы. Имена таких объектов имеют вид \Device\Harddisk*X*\DR*X*, где *X* — номер диска. Для идентификации разделов и создания объектов «устройство», представляющих эти разделы, драйвер класса дисков в Windows 2000 использует функцию *IoReadPartitionTable* диспетчера ввода-вывода, а в Windows XP и Windows Server 2003 — функцию *IoReadParititionTableEx*. Драйвер класса дисков вызывает одну из этих функций для каждого диска, представленного минипорт-драйвером драйверу класса на ранних стадиях загрузки системы. А функция инициирует дисковый ввод-вывод на уровне секторов, поддерживаемый драйвером класса, порт- и минипорт-драйверами, для считывания MBR- или GPT-таблицы разделов (об этом мы расскажем позже) и для формирования внутреннего представления жестких разделов диска. Драйвер класса дисков создает объекты «устройство», представляющие все главные разделы (в том числе логические диски внутри дополнительных разделов), которые этот драйвер получает от *IoReadPartitionTable* или *IoReadParititionTableEx*. Вот пример имени объекта раздела:

```
\Device\Harddisk0\DP(1)0x7e000-0x7ff50c00+2
```

Это имя идентифицирует первый раздел первого диска системы. Два первых шестнадцатеричных числа (0x7e000 и 0x7ff50c00) определяют начало и длину раздела, а последнее число — внутренний идентификатор, назначенный драйвером класса.

Для совместимости с приложениями, использующими правила именования, принятые в Windows NT 4, драйвер класса дисков формирует для имен в формате Windows NT 4 символьные ссылки на объекты «устройство», созданные драйвером. Например, драйвер класса создает ссылки \Device\Hard-

disk0\Partition0 на \Device\Harddisk0\DR0 и \Device\Harddisk0\Partition1 на объект «устройство» первого раздела первого диска. В Windows драйвер класса создает такие же символьные ссылки, представляющие физические диски, созданные в системах под управлением Windows NT 4. Так, ссылка \??\PhysicalDrive0 указывает на \Device\Harddisk0\DR0. На рис. 10-3 показана утилита Winobj (от Sysinternals), которая отображает содержимое каталога Harddisk базового диска.

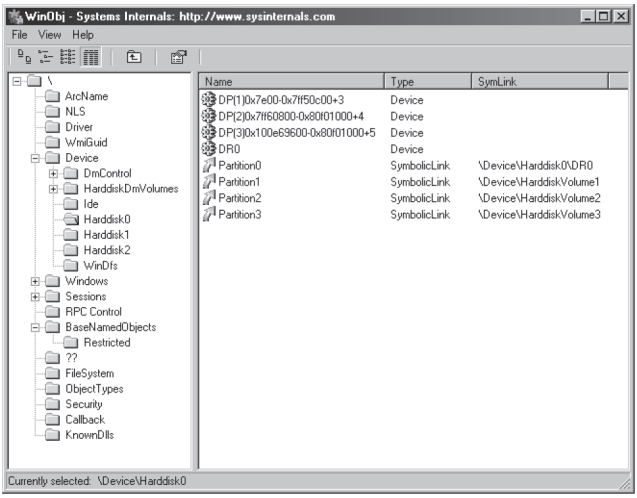


Рис. 10-3. *Окно Winobj, показывающее содержимое каталога Harddisk базового диска*

Как вы уже видели в главе 3, Windows API ничего не знает о пространстве имен диспетчера объектов. Windows резервирует два подкаталога пространства имен, один из которых — подкаталог \Global?? (\?? в Windows 2000). (Другой подкаталог, \BaseNamedObjects, был рассмотрен в главе 3.) В этом подкаталоге объекты «устройство», включая диски, последовательные и параллельные порты, становятся доступными Windows-приложениям. Так как на самом деле объекты дисков находятся в других подкаталогах, для связывания имен в \Global?? с объектами, расположенными в других каталогах пространства имен, Windows использует символьные ссылки. Диспетчер ввода-вывода создает ссылку \Global??\PhysicalDrive*X* для каждого физического диска системы; такая ссылка указывает на \Device\Harddisk*X*\Partition0 (где *X* — числа, начиная с 0). Windows-приложения, напрямую обращающиеся к секторам диска, открывают диск вызовом Windows-функции *CreateFile* и указывают в качестве параметра имя \\.\PhysicalDrive*X* (где *X* — номер диска). Прежде чем передать имя диспетчеру объектов, прикладной уровень Windows преобразует его в \Global??\PhysicalDrive*X*.

Диспетчер разделов

Диспетчер разделов (partition manager), \Windows\System32\Drivers\Partmgr.sys, отвечает за уведомление диспетчера Plug and Play (PnP) о наличии разделов; благодаря этому драйверы диспетчера томов (о них чуть позже) могут получать уведомления о создании и удалении разделов.

Для получения информации о разделах диспетчер разделов действует как функциональный драйвер применительно к объектам дисковых устройств, создаваемых драйвером класса дисков. При загрузке системы он считывает таблицы разделов подключенных дисков (в Windows 2000 через функцию ядра *IoReadPartitionTable*, а в Windows XP и Windows Server 2003 через такую же функцию *IoReadPartitionTableEx*) и сообщает об имеющихся разделах диспетчеру PnP. Драйверы устройств диспетчера томов получают уведомление о разделах управляемых ими дисков и на основании сведений обо всех разделах, из которых состоят тома, определяют объекты «том». Диспетчер разделов отслеживает пакеты запросов на ввод-вывод (I/O request packets, IRP), относящиеся к модификации таблицы разделов, и поэтому может обновлять внутреннюю карту разделов, а затем уведомлять диспетчер PnP о создании и удалении любых разделов.

Управление томами

В Windows введена концепция *базовых* (basic) и *динамических* (dynamic) дисков. Диски с разбиением на разделы исключительно по схеме MBR или GPT в Windows называются базовыми. Поддержка динамических дисков впервые появилась в Windows 2000; они реализуют более гибкую схему разбиения на разделы, чем базовые. Фундаментальное различие между базовыми и динамическими дисками в том, что последние поддерживают создание составных томов (более производительных и надежных, чем простые тома). По умолчанию Windows управляет всеми дисками как базовыми — динамические диски надо создавать вручную или преобразованием из существующих базовых (если на них достаточно свободного места). Но если вам не нужна функциональность составных томов, Microsoft рекомендует использовать именно базовые диски.

ПРИМЕЧАНИЕ Составные тома поддерживаются и на базовых дисках, но только если эти тома переносятся из Windows NT 4 (исключение составляет Windows Server 2003, которая в принципе не поддерживает составные тома на базовых дисках.) На портативных компьютерах — в силу ряда причин, в том числе из-за наличия только одного жесткого диска, который не предназначен для переноса между компьютерами, — Windows использует исключительно базовые диски. Кроме того, динамическими могут быть лишь фиксированные диски. Диски, подключенные к шинам IEEE 1394 или USB, а также диски, совместно используемые серверным кластером, всегда являются базовыми.

Эволюция управления внешней памятью

Эволюция управления внешней памятью началась с MS-DOS, первой операционной системы Microsoft. Когда емкость жестких дисков увеличилась, в MS-DOS нужно было ввести соответствующую поддержку. Поэтому первым шагом Microsoft стала организация в MS-DOS поддержки нескольких разделов, или логических дисков, на одном физическом диске. MS-DOS позволяла форматировать разделы с использованием различных файловых систем (FAT12 или FAT16) и назначать каждому разделу свою букву диска. Количество и размер разделов, которые можно было создавать в MS-DOS версий 3 и 4, были жестко ограничены, но уже в MS-DOS 5 схема разбиения на разделы стала вполне зрелой. MS-DOS 5 умела разбивать диски на любое число разделов произвольного размера.

Windows NT унаследовала схему разбиения жестких дисков на разделы, созданную для MS-DOS. Сделано это было из двух соображений: для совместимости с MS-DOS и Windows 3.*x*, а также для того, чтобы команда разработчиков Windows NT могла опереться на проверенные средства управления дисками. Базовые концепции MS-DOS, относящиеся к разбиению дисков на разделы, в Windows NT были расширены для поддержки функций управления внешней памятью, необходимых операционной системе корпоративного класса, в частности для поддержки перекрытия дисков (disk spanning) и большей отказоустойчивости. В первой версии Windows NT, Windows NT 3.1, системные администраторы могли создавать тома, состоящие из нескольких разделов, что позволяло формировать тома большого размера из разделов нескольких физических дисков, а также повышать отказоустойчивость дисковой подсистемы за счет избыточности данных, организуемой программными средствами.

Хотя поддержка разбиения дисков на разделы по схеме MS-DOS в версиях Windows NT, предшествовавших Windows 2000, была достаточно гибкой для многих задач управления внешней памятью, у нее все же был ряд недостатков. Один из них в том, что активизация большинства изменений в конфигурации дисков требует перезагрузки системы. Но современные серверы должны непрерывно работать в течение месяцев и даже лет, поэтому любая перезагрузка, даже плановая, крайне нежелательна. Другой недостаток связан с тем, что в Windows NT 4 информация о конфигурации томов, состоящих из нескольких разделов и созданных на основе MS-DOS-разделов, хранится в реестре. Это крайне затрудняет перенос конфигурационной информации при перемещении дисков между системами, а при переустановке операционной системы возможна и потеря этой информации. Наконец, требование назначать каждому тому уникальные буквы дисков из диапазона A–Z уже давно досаждало пользователям операционных сис-

тем Microsoft, ограничивая возможное количество локальных и подключенных сетевых томов.

Windows поддерживает три типа разбиения на разделы, которые позволяют преодолевать упомянутые ограничения: MBR (Master Boot Record), GPT (GUID Partition Table) и LDM (Logical Disk Manager).

Базовые диски

В этом разделе описываются два типа разбиения на разделы — MBR и GPT, используемые Windows для определения томов на базовых дисках, — а также драйвер диспетчера томов (FtDisk), представляющий тома драйверам файловых систем. Если диспетчер дисков в Windows 2000 рекомендовал вам делать любой неразмеченный диск динамическим, то Windows XP и Windows Server 2003 автоматически определяют все диски как базовые.

Разбиение на разделы по схеме MBR

Одно из требований к формату разбиения на разделы в Windows диктуется стандартными реализациями BIOS в системах: первый сектор основного диска должен содержать *главную загрузочную запись* (Master Boot Record, MBR). При запуске компьютера на базе x86-процессора BIOS считывает MBR и интерпретирует ее часть как исполняемый код. Выполнив предварительное конфигурирование оборудования, BIOS передает управление исполняемому коду в MBR для инициации процесса загрузки операционной системы.

В операционных системах Microsoft, включая Windows, MBR также содержит таблицу разделов. *Таблица разделов* состоит из четырех элементов, определяющих местонахождение на диске четырех главных разделов. В этой таблице указываются и типы разделов (которые определяют, какую файловую систему включает тот или иной раздел). Существует множество предопределенных типов разделов, например для FAT32 и NTFS. Раздел особого типа, *дополнительный* (extended partition), содержит еще одну MBR с собственной таблицей разделов. Эквивалент главного раздела в дополнительном называется *логическим диском*. Применение дополнительных разделов позволяет операционным системам Microsoft создавать любое количество разделов на диске (а не четыре на один диск).

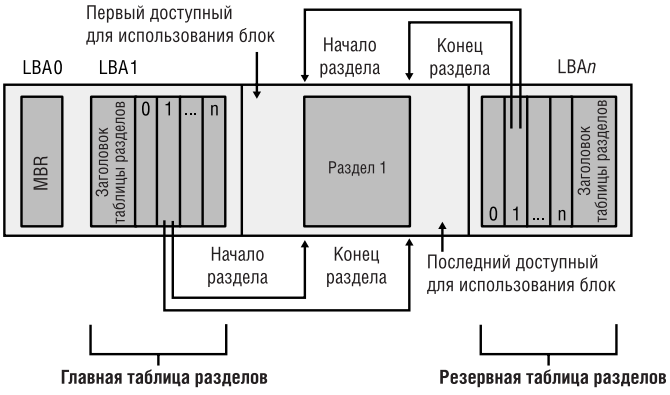
Отличие главного раздела от логических дисков становится очевидным при загрузке Windows. Один из главных разделов основного жесткого диска должен быть помечен системой как активный. Код Windows, записываемый в MBR, загружает в память код первого сектора активного раздела (системного тома) и передает ему управление. Первый сектор такого раздела называется загрузочным. Кроме того, как уже говорилось в главе 4, у каждого раздела, отформатированного с использованием определенной файловой системы, имеется свой загрузочный сектор, который хранит информацию о структуре файловой системы данного раздела.

Разбиение на разделы по схеме GPT

В рамках инициативы, направленной на создание стандартизированной и расширяемой платформы микрокода, которую операционные системы могли бы использовать в процессе своей загрузки, корпорация Intel разработала спецификацию EFI (Extensible Firmware Interface). EFI включает среду операционной мини-системы, реализуемую в виде микрокода, который, как правило, зашивается в ПЗУ. Эта среда используется операционной системой на ранних этапах для загрузки системных диагностических процедур и загрузочного кода. Первый процессор, поддерживающий EFI, — Intel IA64, поэтому версии Windows для IA64 используют EFI, но при желании позволяют выбрать и схему MBR. Детальное описание EFI см. по ссылке *http://developer.intel.com/technology/efi*.

EFI определяет схему разбиения на разделы — таблицу разделов GUID (GUID Partition Table, GPT), которая должна устранить некоторые недостатки схемы разбиения MBR. Например, адреса секторов, используемых структурами разделов GPT, вместо 32-разрядных стали 64-разрядными. 32-разрядные адреса обеспечивают доступ к 2 Тб памяти, но GPT разработана с прицелом на обозримое будущее. Среди прочих преимуществ GPT стоит отметить применение контрольных сумм CRC (cyclic redundancy checksums) для поддержания целостности таблицы разделов, а также резервное копирование таблицы разделов. GPT получила такое название из-за того, что кроме 36-байтового Unicode-имени она назначает каждому разделу свой GUID.

На рис. 10-4 показан пример структуры раздела GPT. Как и в MBR-схеме, первый сектор GPT-диска содержит главную загрузочную запись, которая защищает этот диск от доступа операционных систем, не поддерживающих GPT. Но во втором и последнем секторах диска хранятся заголовки таблицы разделов GPT, а сама таблица размещается сразу за вторым сектором и перед последним сектором. Поддержка расширяемого списка разделов исключает необходимость во вложенных разделах, используемых в схеме MBR.



Примечание: LBA - логический адрес блока

Рис. 10-4. *Пример структуры GPT-раздела*

ПРИМЕЧАНИЕ Windows не поддерживает создание составных томов на базовых дисках, и новый раздел базового диска эквивалентен тому. Именно поэтому в оснастке Disk Management (Управление дисками) консоли MMC для обозначения тома, созданного на базовом диске, используется термин «раздел» (partition).

Диспетчер томов на базовых дисках

Драйвер FtDisk (\Windows\System32\Drivers\Ftdisk.sys) создает объекты «устройство» дисков для представления томов на базовых дисках и играет основную роль в управлении всеми томами на базовых дисках, включая простые тома. Для каждого тома FtDisk создает объект «устройство» вида \Device \HarddiskVolume*X*, где *X* — число, которое идентифицирует том и начинается с 1.

На самом деле FtDisk является драйвером шины, поскольку отвечает за перечисление базовых дисков для обнаружения базовых томов и за оповещение о них диспетчера PnP. Определяя существующие разделы на базовых дисках, FtDisk использует диспетчер PnP и драйвер диспетчера разделов (Partmgr.sys). Диспетчер разделов регистрируется у диспетчера PnP, поэтому Windows может уведомить диспетчер разделов о том, что драйвер класса диска создал объект «устройство» раздела. Диспетчер разделов информирует FtDisk о новых объектах раздела через закрытый интерфейс и создает объекты «устройство» фильтра (filter device objects), которые потом подключает к объектам «устройство» разделов. При наличии объектов «устройство» фильтра Windows посылает диспетчеру разделов уведомление всякий раз, когда удаляется объект «устройство» раздела, что позволяет диспетчеру разделов обновлять информацию FtDisk. Драйвер класса дисков удаляет объект раздела при удалении раздела с помощью оснастки Disk Management консоли MMC. Получив сведения о наличии разделов, FtDisk на основе информации о конфигурации базовых дисков определяет соответствие между разделами и томами, а затем создает объекты «устройство» томов.

Далее Windows создает в каталоге \Global?? (\?? в Windows 2000) диспетчера объектов символьные ссылки, указывающие на объекты томов, созданные FtDisk. Когда система или приложение впервые обращается к тому, Windows монтирует этот том, что позволяет драйверам файловых систем распознать и захватить во владение тома, отформатированные для поддерживаемых ими файловых систем (о монтировании см. раздел «Монтирование томов» далее в этой главе).

Динамические диски

Мы уже упоминали, что динамические диски в Windows нужны для создания составных томов. За поддержку динамических дисков отвечает подсистема диспетчера логических дисков (Logical Disk Manager, LDM), состоящая из компонентов пользовательского режима и драйверов устройств. Microsoft лицензирует LDM у компании VERITAS Software, которая изначально разработала технологию LDM для UNIX-систем. Тесно сотрудничая с Microsoft,

VERITAS перенесла LDM в Windows, благодаря чему эта операционная система получила более отказоустойчивую схему разбиения на разделы и средства поддержки составных томов. Главное отличие схемы разбиения на разделы LDM в том, что LDM поддерживает одну унифицированную базу данных, где хранится информация о разделах на всех динамических дисках системы, в том числе сведения о конфигурации составных томов.

ПРИМЕЧАНИЕ UNIX-версия LDM поддерживает и *дисковые группы* (disk groups): все динамические диски, включаемые системой в группу, используют общую базу данных. Однако коммерческое программное обеспечение VERITAS для управления логическими дисками в Windows поддерживает создание только одной дисковой группы.

База данных LDM

База данных LDM размещается в зарезервированном пространстве (размером 1 Мб) в конце каждого динамического диска. Именно поэтому Windows требует свободное место в конце базового диска при его преобразовании в динамический. База данных LDM состоит из четырех областей, показанных на рис. 10-5: сектора заголовка, называемого в LDM «Private Header», таблицы оглавления, записей базы данных и журнала транзакций (пятый раздел на рис. 10-5 — просто зеркальная копия Private Header). Сектор Private Header размещается за 1 Мб до конца динамического диска и является границей базы данных. Работая с Windows, вы быстро заметите, что для идентификации практически всех объектов в ней используются GUID, и диски не составляют исключения. GUID — это 128-битное число, применяемое различными компонентами Windows для уникальной идентификации объектов. LDM назначает GUID каждому динамическому диску, а сектор Private Header регистрирует GUID динамического диска, на котором он находится, поэтому данные в Private Header относятся исключительно к конкретному диску. Private Header также хранит указатель на начало таблицы оглавления базы данных и имя дисковой группы, которое формируется конкатенацией имени компьютера и строки Dg0 (если имя компьютера — Daryl, то имя дисковой группы — DarylDg0). (Как уже говорилось, LDM в Windows поддерживает только одну дисковую группу, поэтому ее имя всегда оканчивается на Dg0.) Для большей надежности LDM поддерживает копию Private Header в последнем секторе диска.

Таблица оглавления занимает 16 секторов и содержит информацию о структуре базы данных. Область записей базы данных LDM начинается с сектора заголовка записей базы данных сразу за таблицей оглавления. В этом секторе хранится информация об области записей базы данных, включая число присутствующих в ней записей, имя и GUID дисковой группы, к которой относится база данных, и идентификатор последовательности, используемый LDM для создания следующего элемента в базе данных. Секторы, следующие за сектором заголовка записей, содержат записи фиксированного размера (по 128 байтов) с описанием разделов и томов дисковой группы.

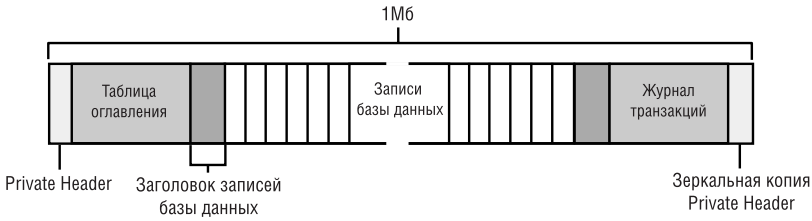


Рис. 10-5. *Структура базы данных LDM*

Элементы базы данных могут быть четырех типов: раздел (partition), диск (disk), компонент (component) и том (volume). Типы элементов определяют три уровня описания томов. LDM связывает элементы с помощью внутренних идентификаторов объектов. На самом нижнем уровне *элементы разделов* (partition entries) описывают нежесткие разделы, которые являются непрерывными областями на диске; идентификаторы, хранящиеся в элементе раздела, связывают его с элементами компонентов и дисков. *Элемент диска* (disk entry) представляет динамический диск в составе группы и включает его GUID. *Элемент компонента* (component entry) служит связующим звеном между одним или несколькими элементами разделов и элементом тома, с которым сопоставлен каждый раздел. *Элемент тома* хранит GUID этого тома, его суммарный размер, информацию о состоянии и букву диска. Элементы дисков, размер которых превышает размер одной записи, занимают несколько записей; элементы разделов, компонентов и томов редко занимают больше одной записи.

Простой том в LDM описывается тремя элементами: раздела, компонента и тома. Ниже показано содержимое простой базы данных LDM, которая определяет один том размером 200 Мб, состоящий из одного раздела.

```
Disk Entry      Volume Entry    Component Entry    Partition Entry
Name: Disk1     Name: Volume1   Name: Volume1-01   Name: Disk1-01
GUID: XXX-XX... ID: 0x408       ID: 0x409          ID: 0x407
Disk ID: 0x404  State: ACTIVE   Parent ID: 0x408   Parent ID: 0x409
                Size: 200MB                        Disk ID: 0x404
                GUID: XXX-XX...                    Start: 300MB
                Drive Hint: H:                     Size: 200MB
```

Элемент раздела описывает область на диске, отведенную тому, элемент компонента связывает элемент раздела с элементом тома, а элемент тома содержит GIUD, используемый Windows на внутреннем уровне для идентификации тома. Для составных томов требуется более трех элементов. Например, чередующийся том (о них — позже) состоит минимум из двух элементов разделов, элемента компонента и элемента тома. Единственный том, который включает более одного элемента компонента, — зеркальный. Зеркальные тома включают два элемента компонентов, каждый из которых представляет половину зеркального тома. Когда вы разбиваете зеркальный том на разделы, LDM может разделить его на уровне компонентов, создав два тома, в каждом из которых будет по одному элементу компонента.

Последняя область базы данных LDM отведена под журнал транзакций. Она состоит из нескольких секторов, предназначенных для хранения резервной копии информации базы данных в процессе ее изменения. Такая схема гарантирует целостность базы данных даже в случае краха системы или сбоя электропитания, поскольку LDM может восстановить согласованное состояние базы данных на основе журнала транзакций.

ЭКСПЕРИМЕНТ: просмотр базы данных LDM с помощью LDMDump

Утилита LDMDump (от Sysinternals) позволяет получить детальную информацию о содержимом базы данных LDM. Она принимает номер диска в качестве аргумента командной строки. Выводимая ею информация занимает несколько экранов, поэтому ее следует перенаправить в файл для последующего просмотра в текстовом редакторе (например: *ldmdump /d0 > disk.txt*). Ниже даны фрагменты выходной информации LDMDump. Первым показывается заголовок базы данных LDM, затем записи этой базы данных, которые описывают 4-гигабайтный диск с простым томом размером в 4 Гб. Элемент тома базы данных обозначен как Volume1. В конце вывода LDMDump перечисляет нежесткие разделы (soft partitions) и определения томов, найденные в базе данных.

```
C:\>ldump /d0
Logical Disk Manager Configuration Dump v1.03
Copyright (C) 2000-2002 Mark Russinovich

PRIVATE HEAD:
Signature          : PRIVHEAD
Version            : 2.11
Disk Id            : b78e4169-2e39-4a57-a98b-afd7131392f9
Host Id            : 1b77da20-c717-11d0-a5be-00a0c91db73c
Disk Group Id      : 2ea3c400-c92a-4172-9286-de46dda1098a
Disk Group Name    : Vmware03Dg0
Logical disk start : 3F
Logical disk size  : 7FF54B (4094 MB)
Configuration start: 7FF7E0
Configuration size : 800 (1 MB)
Number of TOCs     : 1
TOC size           : 7FE (1023 KB)
Number of Configs  : 1
Config size        : 5AC (726 KB)
Number of Logs     : 1
Log size           : DC (110 KB)

TOC 0:
Signature          : TOCBLOCK
Sequence           : 0x9
Config bitmap start: 0x11
Config bitmap size : 0x5AC
```

```
...
VBLK DATABASE:
0x000004: [000004] <Disk>
         Name        : Disk1
         Object Id   : 0x0403
         Disk Id     : b78e4169-2e39-4a57-a98b-afd7131392f9

0x000005: [000002] <DiskGroup>
         Name        : Vmware03Dg0
         Object Id   : 0x0401
         GUID        : 2ea3c400-c92a-4172-9286-de46dda1098a
0x000006: [000006] <Volume>
         Name        : Volume1
         Object Id   : 0x0406
         Volume state: ACTIVE
         Size        : 0x007FB68A (4086 MB)
         GUID        : e6ee6edc-d1ba-11d8-813e-806e6f6e6963
         Drive Hint  : C:

...
PARTITION LAYOUT:
Disk Disk1:
        Disk1-01 Offset: 0x00000000 Length: 0x007FB68A (4086 MB)
VOLUME DEFINITIONS: Volume1 Size: 0x007FB68A (4086 MB)
    Volume1-01 -
      Disk1-01   VolumeOffset: 0x00000000 Offset: 0x00000000
                 Length: 0x007FB68A
```

Схемы разбиения на разделы LDM и GPT или MBR

Одно из первых действий при установке Windows на компьютер — создание раздела на основном физическом диске системы. В этом разделе Windows определяет системный том для хранения файлов, нужных на ранних этапах процесса загрузки. Кроме того, Windows Setup требует создать раздел для загрузочного тома, в который она запишет системные файлы Windows и где будет создан системный каталог (\Windows). Системный том можно сделать и загрузочным — тогда вам не понадобится создавать новый раздел для загрузочного тома. Терминология, используемая Microsoft для системного и загрузочного томов, может сбить с толку. Системным считается том, на который Windows помещает загрузочные файлы, включая загрузчик (Ntldr) и Ntdetect, а загрузочным — том, на котором Windows хранит основные системные файлы вроде Ntoskrnl.exe.

Хотя данные о разбиении динамического диска на разделы находятся в базе данных, LDM реализует и таблицу разделов в стиле MBR или GPT, чтобы загрузочный код Windows мог найти системный и загрузочный тома на динамических дисках. (Например, Ntldr и микрокод IA64 ничего не знают о LDM-разделах.) Если диск содержит системный и/или загрузочный тома, они

описываются в таблице разделов в стиле MBR или GPT. В ином случае один раздел охватывает всю доступную для использования область диска. LDM помечает его как раздел типа «LDM»; поддержка разделов этого типа впервые появилась в Windows 2000. В пространстве, определенном в стиле MBR или GPT, LDM создает разделы, организуемые базой данных LDM.

Еще одна причина, по которой LDM создает таблицу разделов в стиле MBR или GPT, — чтобы унаследованные утилиты обслуживания дисков, включая работающие в средах с двухвариантной загрузкой, не решили, будто на динамическом диске не определены разделы.

Поскольку LDM-разделы не описываются таблицей разделов в стиле MBR или GPT, они называются *нежесткими* (soft partitions), а разделы в стиле MBR или GPT — *жесткими* (hard partitions). Рис. 10-6 иллюстрирует структуру динамического диска, размеченного в стиле MBR.

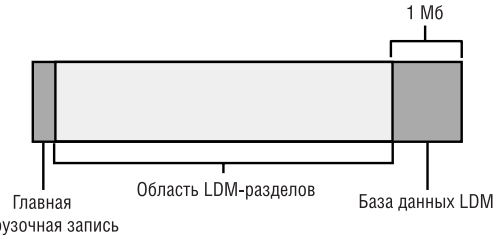


Рис. 10-6. *Внутренняя организация динамического диска*

Диспетчер томов на динамических дисках

DLL оснастки Disk Management (DMDiskManager в Windows\System32\Dmdskmgr.dll), показанной на рис. 10-7, использует DMAdmin, службу LDM Disk Administrator (Windows\System32\Dmadmin.exe) для создания базы данных LDM и изменения ее содержимого. Когда вы запускаете оснастку Disk Management, в память загружается DMDiskManager, который запускает DMAdmin, если она еще не выполняется. DMAdmin считывает с каждого диска базу данных LDM и возвращает полученную информацию DMDiskManager. Если DMAdmin обнаруживает базу данных из дисковой группы другого компьютера, то отмечает, что эти тома находятся на удаленном диске, и позволяет импортировать их в базу данных текущего компьютера, если вы хотите их использовать. При изменении конфигурации динамических дисков DMDiskManager информирует DMAdmin о внесенных изменениях, и DMAdmin обновляет свою копию базы данных в памяти. Зафиксировав изменения, DMAdmin передает обновленную базу данных DMIO, драйверу устройства Dmio.sys. DMIO — эквивалент FtDisk для динамических дисков, так что он управляет доступом к базе данных на диске и создает объекты «устройство», представляющие тома на динамических дисках. Когда вы закрываете оснастку Disk Management, DMDiskManager останавливает и выгружает службу DMAdmin.

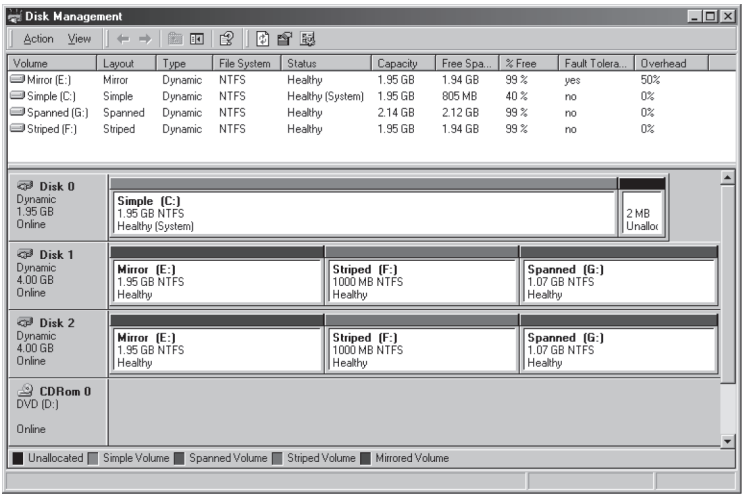


Рис. 10-7. *Оснастка Disk Management (Управление дисками)*

DMIO не знает, как интерпретировать базу данных, которой он управляет. За интерпретацию базы данных отвечают DMConfig (Windows\System32\Dmconfig.dll), загружаемый DMAdmin, и DMBoot (Dmboot.sys), еще один драйвер устройства. DMConfig известно, как считывать и обновлять базу данных, а DMBoot — только как ее считывать. DMBoot загружается при загрузке системы, если другой драйвер LDM, DMLoad (Dmload.sys), обнаруживает в системе минимум один динамический диск. DMLoad определяет наличие динамических дисков, запрашивая DMIO. Если в системе есть хотя бы один динамический диск, DMLoad запускает DMBoot, который сканирует базу данных LDM. DMBoot информирует DMIO о составе каждого найденного им тома, что позволяет DMIO создать объекты «устройство» для представления томов. Закончив сканирование, DMBoot тут же выгружается из памяти. Поскольку в DMIO не заложена логика для интерпретации базы данных, его размер относительно невелик. Это несомненное преимущество, так как DMIO постоянно находится в памяти.

Как и FtDisk, DMIO является драйвером шины и создает объект «устройство» для каждого тома динамического диска, присваивая ему имя в виде \Device\HarddiskDmVolumes\PhysicalDmVolumes\BlockVolume*X*, где *X* — идентификатор тома, назначаемый DMIO. Кроме того, DMIO создает объект «устройство» с именем \Device\HarddiskDmVolumes\PhysicalDmVolumes\RawVolume*X*, который представляет необработанный (неструктурированный) ввод-вывод на томе. Объекты «устройство», созданные DMIO в системе с тремя томами на динамических дисках, показаны на рис. 10-8. DMIO также создает в пространстве имен диспетчера объектов символьные ссылки на все тома, в том числе по одной ссылке для каждого тома в виде \Device\HarddiskDmVolumes*ComputerName*Dg0\Volume*Y*. DMIO заменяет *ComputerName* именем компьютера, а *Y* — идентификатором тома (который отличается от внутреннего идентификатора, назначаемого драйвером DMIO объектам

«устройство»). Эти ссылки указывают на объекты блочных устройств в каталоге PhysicalDmVolumes.

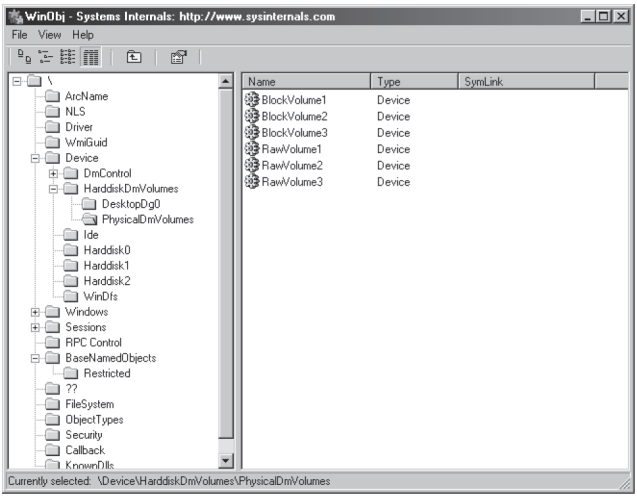


Рис. 10-8. *Объекты «устройство» драйвера DMIO*

ПРИМЕЧАНИЕ В Windows 2000 имеется еще один драйвер — DiskPerf (\Windows\System32\Drivers\Diskperf.sys); он подключается к объектам «устройство», представляющим физические устройства (например, к \Device\Harddisk0\Partition0), что позволяет ему отслеживать запросы ввода-вывода, адресованные дискам, и генерировать статистические данные для соответствующих счетчиков оснастки Performance. В Windows XP и Windows Server 2003 функциональность DiskPerf реализована в драйвере диспетчера разделов, так как он уже фильтрует объекты дисковых устройств для поддержки своих основных функций, о которых мы рассказывали ранее.

Управление составными томами

FtDisk и DMIO отвечают за представление томов, управляемых драйверами файловой системы, и за перенаправление ввода-вывода, адресованного томам, в нижележащие разделы, составляющие тома. В случае простых томов диспетчер томов преобразует смещение в томе в смещение на диске, суммируя смещение в томе со смещением тома от начала диска.

Составные тома более сложны, поскольку составляющие их разделы могут быть несмежными и даже находиться на разных дисках. Некоторые типы составных томов используют избыточность данных и требуют еще более сложной трансляции. Таким образом, FtDisk и DMIO должны обрабатывать все запросы ввода-вывода, адресованные составным томам, и определять, на какие разделы следует направлять тот или иной запрос.

В Windows поддерживаются следующие типы составных томов:

- перекрытые (spanned volumes);
- зеркальные (mirrored volumes);
- чередующиеся (striped volumes);
- RAID-5.

Рассмотрев конфигурацию разделов составных томов и логические операции для каждого типа составных томов, мы обсудим, как драйверы FtDisk и DMIO обрабатывают IRP, посылаемые драйвером файловой системы составному тому. Термин «диспетчер томов» при объяснении составных томов используется для обозначения DMIO, поскольку, как уже говорилось в этой главе, FtDisk поддерживает лишь те составные тома, которые были перенесены из NT 4.

Перекрытые тома

Перекрытый том — единый логический том, состоящий из нескольких (до 32) свободных разделов на одном или нескольких дисках. Оснастка Disk Management (Управление дисками) консоли MMC объединяет разделы в перекрытый том, который затем можно отформатировать для любой файловой системы, поддерживаемой Windows . На рис. 10-9 показан 100-мегабайтный перекрытый том с именем D:, созданный из последней трети первого диска и первой трети второго диска. В Windows NT 4 перекрытые тома назывались *наборами томов* (volume sets).

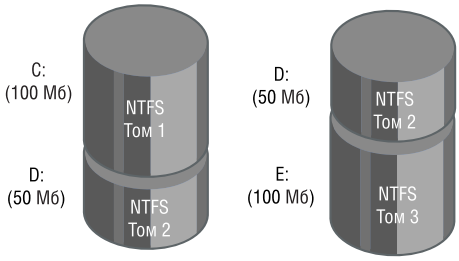


Рис. 10-9. *Перекрытый том*

Перекрытый том удобен для объединения небольших областей свободного дискового пространства в единый том большего объема или для создания из нескольких малых дисков одного большого тома. Если перекрытый том отформатирован для NTFS, его можно расширять, добавляя другие свободные области или диски, и это не влияет на данные, уже хранящиеся на томе. Расширяемость — одно из самых крупных преимуществ описания всех данных на томе NTFS как единого файла. Размер логического тома NTFS может динамически увеличиваться, поскольку битовая карта, регистрирующая состояние тома, — не более чем еще один файл, файл битовой карты. Этот файл может быть расширен для учета пространства, добавляемого в том. С другой стороны, динамическое расширение тома FAT потребовало бы расширения самой FAT, что привело бы к смещению всех данных на диске.

Диспетчер томов скрывает физическую конфигурацию дисков от файловых систем, установленных в Windows. Например, на рис. 10-9 файловая система NTFS рассматривает том D: как обыкновенный 100-мегабайтный том. Чтобы определить свободное пространство на этом томе, NTFS обращается к своей битовой карте. Далее она вызывает диспетчер томов для чтения или записи данных с конкретного смещения в байтах относительно начала тома. Диспетчер томов последовательно нумерует физические сектора перекрытого тома от первой области первого диска до последней области последнего диска. Он определяет, какой физический сектор и на каком диске соответствует указанному смещению.

Чередующиеся тома

Чередующийся том — группа разделов (до 32), каждый из которых размещается на отдельном диске и объединяется в один логический том. Чередующиеся тома также называются томами RAID уровня 0, или томами RAID-0. На рис. 10-10 показан чередующийся том, состоящий из трех разделов, каждый из которых находится на отдельном диске. (Раздел чередующегося тома не обязательно занимает весь диск; единственное ограничение — все разделы на каждом диске должны быть одинаковы.)

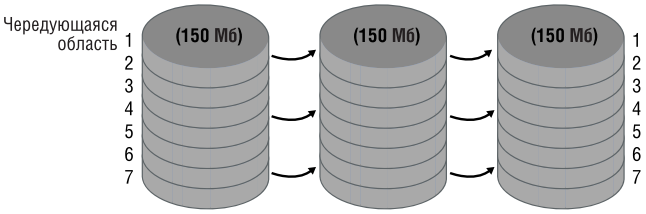


Рис. 10-10. *Чередующийся том*

Файловой системе этот чередующийся том кажется обычным 450-мегабайтным томом, но диспетчер томов оптимизирует хранение и выборку данных на таком томе, распределяя их между физическими дисками. Диспетчер томов обращается к физическим секторам дисков так, как показано на рис. 10-11.

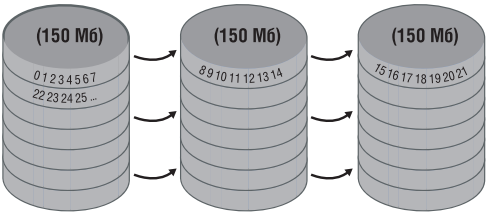


Рис. 10-11. *Логическая нумерация физических секторов в чередующихся томах*

Поскольку каждая чередующаяся область занимает всего 64 Кб (это значение выбрано для того, чтобы отдельные операции чтения и записи не требовали обращения сразу к двум дискам), данные более-менее равномерно распределяются между дисками. Таким образом, чередование увеличивает

вероятность того, что несколько одновременно ожидающих выполнения операций ввода-вывода потребуют доступа к разным дискам. А поскольку к данным на всех трех дисках можно обращаться одновременно, время задержки при дисковом вводе-выводе часто снижается, особенно в условиях высокой нагрузки.

Чередующиеся тома упрощают управление томами и позволяют распределять нагрузку между несколькими дисками, значительно ускоряя ввод-вывод. Но это не обеспечивает восстановления данных в случае сбоя диска. В связи с этим диспетчер томов реализует три механизма избыточности: зеркальные тома, тома RAID-5 и замена секторов (последний механизм описывается в главе 12). Эти возможности можно задействовать через оснастку Disk Management.

Зеркальные тома

В *зеркальном томе* содержимое раздела на одном диске дублируется в разделе равного размера на другом диске (рис. 10-12). Такие тома иногда называют томами RAID уровня 1, или томами RAID-1.

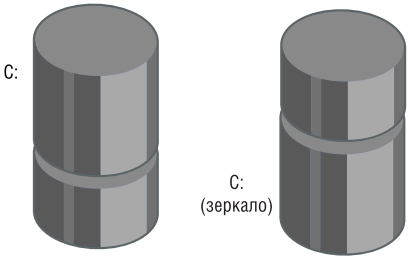


Рис. 10-12. *Зеркальный том*

Когда программа что-то записывает на диск C:, диспетчер томов помещает те же данные в идентичный участок на зеркальный раздел. Если первый диск (или часть данных на нем) окажется поврежденной из-за аппаратного или программного сбоя, диспетчер томов автоматически обратится за нужными данными к зеркальному разделу. Зеркальный том можно отформатировать для любой файловой системы, поддерживаемой Windows. При этом драйверы файловых систем остаются независимыми — зеркалирование никак на них не влияет.

Зеркальные тома способствуют увеличению пропускной способности операций чтения в сильно загруженных системах. При высокой интенсивности ввода-вывода диспетчер томов распределяет операции чтения между первичным и зеркальным разделом (учитывая количество незавершенных запросов ввода-вывода для каждого диска). Две операции чтения могут быть выполнены одновременно, т. е. теоретически вдвое быстрее. При модификации файла приходится вести запись в оба раздела зеркального набора, но запись на диск выполняется асинхронно, и дополнительная операция записи почти не влияет на быстродействие программ пользовательского режима.

Зеркальный том — единственный тип составного тома, допустимого для системного и загрузочного томов. Дело в том, что загрузочный код Windows, включая код MBR и Ntldr, не обладает сложной логикой, необходимой для работы с составными томами. Зеркальные тома составляют исключение, так как загрузочный код воспринимает их как простые тома, считывая данные с той половины зеркального тома, которая помечена как загрузочный или системный диск в таблице разделов MBR. Поскольку загрузочный код не модифицирует данные на диске, он может игнорировать вторую половину зеркального тома.

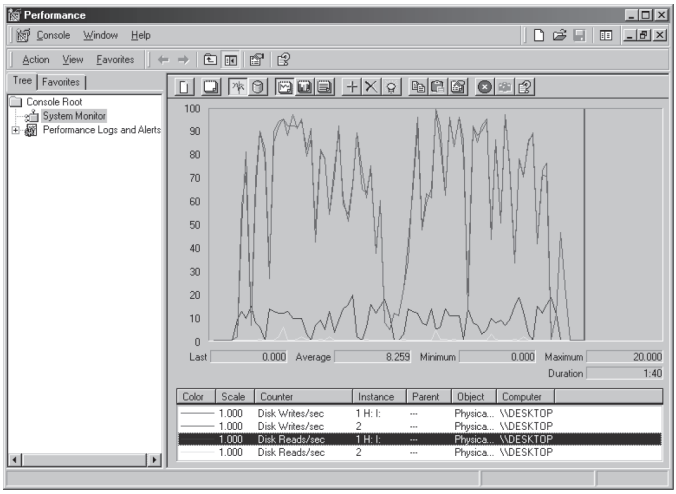
ЭКСПЕРИМЕНТ: наблюдаем за операциями ввода-вывода на зеркальном томе

Используя оснастку Performance (Производительность), вы можете убедиться, что при записи на зеркальные тома данные копируются на оба диска, составляющих зеркальный том (зеркало), а операции чтения, если они не слишком частые, выполняются в основном на одной из половин зеркального тома. Для этого эксперимента вам понадобится система с тремя жесткими дисками под управлением серверной ОС Windows 2000 или Windows Server 2003. Если у вас нет такой системы, пропустите инструкции по подготовке эксперимента и переходите сразу к результатам.

Создайте зеркальный том с помощью оснастки Disk Management.

1. Запустите Computer Management (Управление компьютером), раскройте узел Storage (Запоминающие устройства) и выберите папку Disk Management (Управление дисками) или откройте Disk Management как оснастку консоли MMC.
2. Щелкните правой кнопкой мыши на свободном пространстве диска и выберите команду Create Volume (Создать том).
3. Следуйте инструкциям мастера создания тома, чтобы создать простой том. (Сначала убедитесь, что на другом диске достаточно свободного места для создания тома равного размера.)
4. Щелкните правой кнопкой мыши новый том и из контекстного меню выберите команду Add Mirror (Добавить зеркало).

Создав зеркальный том, запустите оснастку Performance (Производительность) и добавьте счетчики к объекту PhysicalDisk (Физический диск) для каждого экземпляра диска, содержащего раздел зеркального тома. Выберите счетчики Disk Reads/Sec (Обращений чтения с диска/сек) и Disk Writes/Sec (Обращений записи на диск/сек). На третьем диске, не входящем в состав зеркального тома, выберите большой каталог и скопируйте его на зеркальный том. Информация в окне оснастки Performance по мере выполнения операции копирования должна выглядеть примерно так, как показано на иллюстрации.



Две верхних пересекающихся линии представляют графики для значений Disk Writes/Sec по каждому диску, а две нижних — графики для значений Disk Reads/Sec. Как видите, диспетчер томов (в данном случае — DMIO) записывает данные копируемых файлов в обе половины тома, но считывает преимущественно из одной. Это происходит потому, что число незавершенных операций ввода-вывода при копировании невелико и не заставляет диспетчер томов распределять нагрузку по операциям чтения между дисками.

Тома RAID-5

Том RAID-5 — отказоустойчивый вариант обычного чередующегося тома. В томах RAID-5 реализуется RAID уровня 5. Эти тома также называются *чередующимися томами с записью четности* (striped volumes with parity), поскольку они основаны на том же принципе чередования. Отказоустойчивость достигается за счет резервирования эквивалента одного диска для хранения информации о четности для всех чередующихся областей. Том RAID-5 представлен на рис. 10-13.

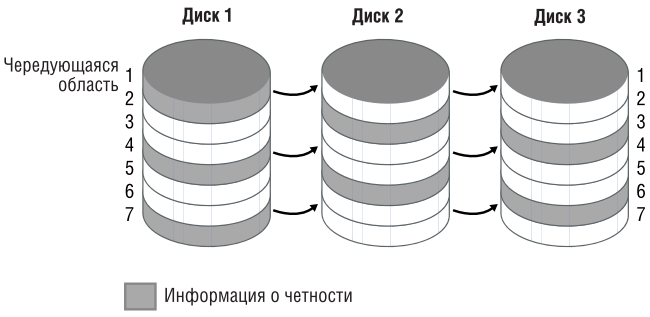


Рис. 10-13. *Том RAID-5*

Как показано на рис. 10-13, информация о четности для чередующейся области 1 хранится на диске 1. Она представляет собой побайтовую логическую сумму (XOR) первых чередующихся областей на дисках 2 и 3. Информация о четности для чередующейся области 2 хранится на диске 2, а для чередующейся области 3 — на диске 3. Такое циклическое размещение информации о четности по дискам представляет собой способ оптимизации ввода-вывода. Всякий раз, когда данные записываются на какой-либо из дисков, байты четности, соответствующие изменяемым байтам, должны быть пересчитаны и перезаписаны. Если бы информация о четности постоянно записывалась на один и тот же диск, он был бы все время занят и мог бы стать узким местом для ввода-вывода.

Восстановление диска после сбоя в томе RAID-5 основывается на простом арифметическом принципе: если в уравнении с n переменными известны значения $n - 1$ переменных, то значение оставшейся переменной можно определить вычитанием. Например, в выражении $x + y = z$, где z обозначает чередующуюся область с четностью, диспетчер томов вычисляет $z - y$, чтобы определить значение x, и $z - x$, чтобы найти y. Диспетчер томов использует сходную логику для восстановления потерянных данных. Если том RAID-5 выходит из строя или данные на одном из его дисков становятся нечитаемыми, диспетчер томов реконструирует отсутствующие данные, используя операцию XOR (побитовое логическое сложение).

В случае сбоя диска 1 на рис. 10-13 содержимое его чередующихся областей 2 и 5 вычисляется побайтовым логическим сложением соответствующих чередующихся областей на диске 3 с областями четности на диске 2. Содержимое чередующихся областей 3 и 6 определяется побайтовым логическим сложением соответствующих областей на диске 2 с областями четности на диске 3. Для организации тома RAID-5 требуется по крайней мере три диска (а точнее, три одинаковых по размеру раздела на трех дисках).

Пространство имен томов

Такой аспект управления внешней памятью, как назначение томам букв дисков, в Windows существенно изменился по сравнению с Windows NT 4. Несмотря на это, Windows поддерживает назначения букв, переносимые при обновлении системы с Windows NT 4 до Windows . Назначенные буквы Windows NT 4 хранит в разделе реестра HKLM\SYSTEM\Disk. После обновления эта информация сохраняется в других местах, специфичных для Windows, и система больше не ссылается на раздел Disk.

Диспетчер монтирования

Диспетчер монтирования, драйвер устройства Mountmgr.sys, назначает буквы томам динамических и базовых дисков, созданных после установки Windows, а также устройствам CD-ROM, приводам гибких дисков и съемным устройствам. Эта операционная система хранит все буквы дисков, назначенные томам, в разделе реестра HKLM\SYSTEM\MountedDevices. Заглянув в этот

раздел, вы увидите параметры с именами вида \??\Volume{*X*} (где *X* — GUID) и \DosDevices\C:. Такие параметры есть у каждого тома, но не всем томам назначены буквы дисков. Пример раздела реестра MountedDevices диспетчера монтирования показан на рис. 10-14. Заметьте, что этот раздел, как и раздел Disk в Windows NT 4, не входит в набор параметров управления и в связи с этим не восстанавливается при загрузке последней удачной конфигурации (см. главу 4).

Данные, которые хранятся в виде параметров реестра для букв и имен томов базовых дисков, представляют собой сигнатуру диска в стиле Windows NT 4 и начальное смещение от первого раздела, сопоставленного с томом. Аналогичные данные для томов динамических дисков включают внутренний GUID тома, используемый DMIO. Когда диспетчер монтирования инициализируется при загрузке системы, он регистрируется в подсистеме поддержки Plug and Play, что позволяет ему в дальнейшем получать уведомления о создании томов драйвером FtDisk или DMIO. Получив такое уведомление, диспетчер монтирования определяет GUID или сигнатуру диска нового тома и использует GUID тома или сигнатуру его диска как критерий для поиска в своей базе данных, отражающей содержимое раздела реестра MountedDevices. Если поиск заканчивается неудачей, диспетчер монтирования запрашивает у FtDisk или DMIO (смотря кто из них создал том) предлагаемую букву для идентификации тома и сохраняет ее в своей базе данных. FtDisk не дает никаких предложений, а DMIO проверяет возможные назначения в элементе тома базы данных.

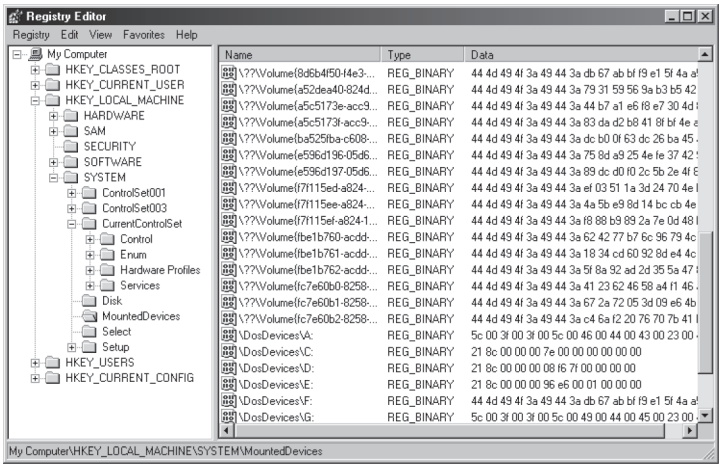


Рис. 10-14. *Смонтированные устройства, перечисленные в разделе реестра, принадлежащем диспетчеру монтирования*

Если диспетчер монтирования не получает никаких предложений, он берет первую свободную букву, назначает ее тому, создает для нее символьную ссылку — например, \Global??\D: в Windows XP и Windows Server 2003 или \??\D: в Windows 2000 — и обновляет раздел реестра MountedDevices. Когда свободных

букв нет, буква не назначается, но создается символьная ссылка \Global??\Volume{*X*}, определяющая GUID нового тома в том случае, если у него еще нет GUID. Этот GUID отличается от GUID томов, используемых DMIO.

Точки монтирования

Точки монтирования (mount points) позволяют связывать тома через каталоги NTFS, делая эти тома доступными без назначения букв дисков. Например, NTFS-каталог C:\Projects может монтировать другой том (NTFS или FAT), содержащий каталоги и файлы ваших проектов. Если в томе ваших проектов есть файл с именем \CurrentProject\Description.txt, то после монтирования путь к нему выглядит как C:\Projects\CurrentProject\Description.txt. Точки монтирования стали возможны благодаря технологии точек повторного разбора (reparse point technology), о которой мы подробно поговорим в главе 12.

Точка повторного разбора — это блок произвольных данных с неким фиксированным заголовком, который Windows сопоставляет с файлом или каталогом NTFS. Приложение или система определяет формат и поведение точки повторного разбора, в том числе значение уникального тэга, который идентифицирует точку повторного разбора, принадлежащую приложению или системе, и указывает размер (до 16 Кб) и смысл данных этой точки. Уникальные тэги хранятся в фиксированном сегменте точек повторного разбора. Любое приложение, реализующее точку повторного разбора, должно предоставлять драйвер фильтра файловой системы, который наблюдает за кодами возврата файловых операций, связанных с повторным разбором и выполняемых на томах NTFS, и предпринимает действия, соответствующие этим кодам. NTFS возвращает код статуса повторного разбора всякий раз, когда обрабатывает файловую операцию применительно к файлу или каталогу, с которым сопоставлена точка повторного разбора.

Драйвер файловой системы NTFS, диспетчер ввода-вывода и диспетчер объектов — каждый из них реализует свою часть функциональности точек повторного разбора. Диспетчер объектов инициирует операции разбора путей файлов, взаимодействуя с драйверами файловых систем через диспетчер ввода-вывода, и должен повторно инициировать операции, для которых диспетчер ввода-вывода возвращает код статуса повторного разбора. Диспетчер ввода-вывода поддерживает модификацию путей, которая может понадобиться точкам монтирования и другим точкам повторного разбора, а драйвер файловой системы NTFS должен связывать данные точек повторного разбора с файлами и каталогами. Поэтому диспетчер ввода-вывода можно рассматривать как драйвер фильтра файловой системы, который поддерживает функциональность повторного разбора для многих точек, определенных Microsoft.

Пример приложения, поддерживающего точки повторного разбора, — система Hierarchical Storage Management (HSM) вроде службы Windows Remote Storage Service (RSS), которая включена в Windows 2000 Server и Windows Server 2003; она использует такие точки для обозначения файлов, перемещаемых администратором в хранилище на ленточных накопителях. Когда пользова-

тель пытается обратиться к автономному файлу, драйвер фильтра HSM обнаруживает код статуса повторного разбора, возвращаемый NTFS, вызывает сервисы пользовательского режима, чтобы получить автономный файл из хранилища, удаляет из файла точку повторного разбора и инициирует повторную попытку выполнения файловой операции. Точно так же точки повторного разбора используются драйвером фильтра RSS (Rsfilter.sys).

Если файл или каталог, для которого диспетчер ввода-вывода получает от NTFS код статуса повторного разбора, не сопоставлен с одной из предопределенных в Windows точек повторного разбора, значит, его точка не обрабатывается ни одним драйвером фильтра. Тогда диспетчер ввода-вывода сообщает диспетчеру объектов об ошибке, которая передается приложению, обратившемуся к этому файлу или каталогу, в виде «file cannot be accessed by the system» («файл недоступен системе»).

Точки монтирования — это точки повторного разбора, в которых имя тома (\Global??\Volume{*X*}) хранится как данные повторного разбора. Назначая или удаляя пути для томов в оснастке Disk Management, вы создаете точки монтирования. Создавать и просматривать точки монтирования можно и с помощью встроенной утилиты командной строки Mountvol.exe (\Windows\System32\Mountvol.exe).

Диспетчер монтирования поддерживает на каждом томе NTFS удаленную базу данных, в которой регистрирует все точки монтирования, определенные для тома. Файл этой базы данных, $MountMgrRemoteDatabase, размещается в корневом каталоге NTFS. При перемещении диска между системами и в средах с двухвариантной загрузкой (различных систем Windows) перемещаются и точки монтирования — благодаря наличию удаленной базы данных диспетчера монтирования. NTFS отслеживает точки монтирования в файле метаданных \$Extend\$Reparse (ни один из файлов метаданных NTFS не доступен приложениям). Поскольку NTFS хранит информацию о точках монтирования в файле метаданных, при соответствующем запросе Windows-приложения Windows может легко перечислить точки монтирования, определенные для тома.

ЭКСПЕРИМЕНТ: рекурсивные точки монтирования

Этот эксперимент с использованием утилиты Filemon (*www.sysinternals.com*) демонстрирует любопытное поведение системы, вызываемое рекурсивной точкой монтирования. Рекурсивной называется точка монтирования, связанная с тем томом, где она находится. Рекурсивное перечисление каталогов, выполняемое на рекурсивной точке монтирования, позволяет наглядно увидеть, как NTFS обрабатывает точки монтирования.

Для создания и просмотра точки монтирования проделайте следующее.

1. Откройте окно командной строки или Windows Explorer и создайте на NTFS-диске каталог с именем \Recurse.

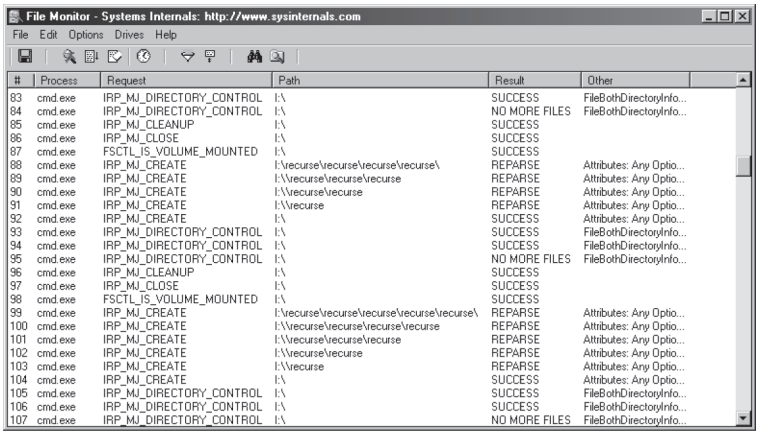
см. след. стр.

2. В оснастке Disk Management (Управление дисками) консоли MMC щелкните том правой кнопкой мыши и выберите из контекстного меню команду Change Drive Letter And Path (Изменить букву диска или путь к диску).
3. В появившемся диалоговом окне введите путь к созданному вами каталогу (например, *I:\Recurse*).
4. Запустите Filemon. В меню Drives оставьте галочку только для тома, на котором создана точка монтирования.

Теперь вы готовы к трассировке рекурсивной точки монтирования. Откройте окно командной строки и введите **dir /s I:\Recurse**. Следите за ссылками на Recurse, регистрируемыми Filemon при трассировке файловых операций. Вы заметите, что сначала идет обращение к I:\Re-curse, затем к I:\Recurse\Recurse и т. д.

Приложение перечисляет каталоги на каждом уровне рекурсии, но всякий раз, когда встречает точку монтирования, оно закапывается все глубже и глубже, пытаясь выполнить очередное перечисление каталогов. NTFS возвращает код статуса повторного разбора, который сигнализирует диспетчеру объектов о необходимости возврата на предыдущий уровень рекурсии и повторной попытки операции. Наконец, вернувшись в корневой каталог, приложение исследует файл или каталог, найденный им при глубокой рекурсии. Приложение никогда не получает код статуса повторного разбора из-за того, что диспетчер объектов сам обрабатывает статусные коды повторного разбора при получении их от NTFS.

Filemon показывает запрос на открытие файла или каталога как IRP_MJ_CREATE, запрос на закрытие файла или каталога — как IRP_MJ_CLOSE, а запрос сведений о каталогах — как IRP_MJ_DIRECTORY _CONTROL, выполняемый с помощью функции *FileBothDirectoryInfo* (см. колонку Other).



Чтобы предотвратить переполнение буферов и вхождение в бесконечный цикл, командный процессор и Windows Explorer останавливают рекурсию по достижении 32-го уровня вложенности или при превышении длины пути в 256 символов — смотря что произойдет быстрее.

Монтирование томов

Тот факт, что Windows присваивает тому букву диска, еще не означает, что этот том содержит данные, организованные в формате файловой системы, известной Windows. Процесс распознавания тома заключается в том, что какая-либо файловая система объявляет раздел своим; первый раз этот процесс происходит при обращении ядра, драйвера устройства или приложения к какому-либо файлу или каталогу в данном томе. После того как драйвер файловой системы уведомляет о взятии на себя ответственности за управление разделом, диспетчер ввода-вывода направляет все адресованные этому тому запросы данному драйверу. Операции монтирования в Windows проходят в три этапа: регистрация драйвера файловой системы, обновление блоков параметров тома (volume parameter blocks, VPB) и запросы на монтирование.

ПРИМЕЧАНИЕ В Windows Server 2003 Enterprise и Datacenter Edition автоматическое монтирование отключено, чтобы не допустить агрессивного монтирования томов, подключенных к сети устройств хранения данных (System Area Network, SAN). Для включения или отключения автоматического монтирования можно использовать утилиту командной строки Diskpart, поставляемую с Windows Server 2003, а для монтирования томов вручную — утилиту Mountvol, также поставляемую с этой системой.

Процесс монтирования курирует диспетчер ввода-вывода, которому известны доступные драйверы файловых систем, поскольку они регистрируются у него при инициализации. Для регистрации драйверов файловых систем на локальных (не сетевых) дисках предназначена функция *IoRegisterFileSystem*, предоставляемая диспетчером ввода-вывода. Когда драйвер файловой системы регистрируется, диспетчер ввода-вывода сохраняет ссылку на драйвер в списке, который используется при операциях монтирования.

Каждый объект «устройство» содержит структуру данных VPB, но диспетчер ввода-вывода обращает внимание только на VPB объектов томов. VPB служит для связи между объектом тома и объектом «устройство», созданным драйвером файловой системы для представления экземпляра файловой системы, смонтированной для данного тома. Если ссылка VPB на файловую систему пуста, значит, том не смонтирован ни одной файловой системой. Диспетчер ввода-вывода проверяет VPB объекта тома всякий раз, когда выполняется API-функция открытия файла или каталога на этом объекте «устройство».

Например, если диспетчер монтирования назначает второму тому системы букву D, он создает символьную ссылку \Global??\D:, представляющую объект \Device\HarddiskVolume2. Windows-приложение, пытающееся открыть

файл \Temp\Test.txt на диске D:, указывает имя D:\Temp\Test.txt, которое подсистема Windows преобразует в \Global??\D:\Temp\Test.txt перед вызовом *NtCreateFile* — процедуры ядра, отвечающей за открытие файла. *NtCreateFile* использует диспетчер объектов для разбора имени, и диспетчер объектов обнаруживает объект «устройство» \Device\HarddiskVolume2 с еще не разрешенным путем \Temp\Test.txt. На этом этапе диспетчер ввода-вывода проверяет, есть ли в VPB объекта \Device\HarddiskVolume2 ссылка на файловую систему. Если нет, диспетчер ввода-вывода выдает зарегистрированному драйверу файловой системы запрос на монтирование, чтобы выяснить, распознает ли он формат монтируемого тома как формат своей файловой системы.

ЭКСПЕРИМЕНТ: просмотр VPB

Увидеть содержимое VPB позволяет команда *!vpb* отладчика ядра. Поскольку на VPB указывает объект «устройство» тома, сначала нужно найти этот объект. Для этого создайте дамп объекта «драйвер» диспетчера томов, найдите объект «устройство», представляющий том, просмотрите его содержимое и вы обнаружите поле VPB.

Если в системе есть динамический диск, используйте команду *!drvobj* применительно к драйверу DMIO, а если нет — применительно к драйверу FtDisk. Вот пример:

```
kd> !drvobj ftdisk
Driver object (818aec50) is for:
 \Driver\Ftdisk
Driver Extension List: (id , addr)

Device Object list:
818a5290  817e96f0  817e98b0  817e9030
818a73b0  818a7810  8182d030
```

Команда *!drvobj* перечисляет адреса объектов «устройство», принадлежащих драйверу. В этом примере таких объектов — семь. Один из них представляет программный интерфейс драйвера устройства, остальные — объекты томов. Поскольку объекты показываются в порядке, обратном порядку их создания, а первым создается объект «устройство» интерфейса драйвера устройства, то первый перечисленный объект «устройство» является объектом тома. Теперь введите команду *!devobj* отладчика ядра, указав в качестве параметра адрес объекта тома.

```
kd> !devobj 818a5290
Device object (818a5290) is for:
 HarddiskVolume6 \Driver\Ftdisk DriverObject 818aec50
Current Irp 00000000 RefCount 3 Type 00000007 Flags 00001050
Vpb 818a5da8 Dacl e1000384 DevExt 818a5348 DevObjExt 818a53f8
  Dope 818a50a8 DevNode 818a5ae8
ExtensionFlags (0xa8000000)  DOE_RAW_FDO, DOE_DESIGNATED_FDO
                             Unknown flags 0x08000000
```

```
AttachedDevice (Upper) 86b52b58 \Driver\VolSnap
Device queue is not busy.
```

Команда *!devobj* показывает поле VPB объекта тома. (Данному объекту «устройство» присвоено имя HarddiskVolume6.) Теперь можно выполнить команду *!vpb*:

```
kd> !vpb 818a5da8
Vpb at 0x818a5da8
Flags: 0x1 mounted
DeviceObject: 0x850dcac0
RealDevice:   0x818a5290
RefCount: 3
Volume Label:      BACKUP
```

В итоге мы выяснили, что объект тома смонтирован драйвером файловой системы, который присвоил ему имя BACKUP. VPB-поле *RealDevice* указывает обратно на объект тома, а поле *DeviceObject* указывает на объект «устройство» смонтированной файловой системы.

По соглашению, драйвер файловой системы при распознавании формата монтируемого тома должен анализировать загрузочную запись тома, хранящуюся в его первом секторе. Загрузочные записи файловых систем Microsoft содержат поле, описывающее тип формата файловой системы. Драйверы файловых систем обычно проверяют это поле и, если оно указывает на поддерживаемый ими формат, анализируют остальную информацию, хранящуюся в загрузочной записи. Эта информация обычно включает имя файловой системы и данные, необходимые для поиска критически важных файлов метаданных тома. Например, NTFS распознает том, только если поля типа и имени определяют именно NTFS, а файлы метаданных, описываемые загрузочной записью, находятся в согласованном состоянии.

Если драйвер файловой системы подтверждает распознавание, диспетчер ввода-вывода заполняет VPB и передает запрос на открытие с оставшейся частью пути (т. е. \Temp\Test.txt) драйверу файловой системы. Последний выполняет запрос, интерпретируя данные в соответствии с форматом своей файловой системы. После того как поля VPB объекта «устройство» тома заполнены нужной информацией, диспетчер ввода-вывода передает все последующие запросы, адресованные данному тому, драйверу смонтированной файловой системы. Если ни один драйвер файловой системы не объявляет этот том своим, владельцем становится Raw — встроенный в Ntoskrnl.exe драйвер файловой системы, который сообщает о неудаче в ответ на любые попытки открыть файл в данном разделе. Рис. 10-15 иллюстрирует упрощенную схему потока ввода-вывода, направляемого на смонтированный том (здесь не показано взаимодействие драйвера файловой системы с диспетчерами кэша и памяти).

Вместо того чтобы загружать все драйверы файловых систем независимо от наличия соответствующих томов, Windows пытается свести к минимуму

нагрузку на память, используя для предварительного распознавания файловой системы суррогатный драйвер File System Recognizer (Windows \System32\Drivers\Fs_rec.sys). Этот драйвер знает о формате каждой файловой системы, поддерживаемой Windows, ровно столько, чтобы суметь проанализировать загрузочную запись и определить, можно ли ее сопоставить с какой-нибудь файловой системой Windows. При загрузке системы File System Recognizer регистрируется как драйвер файловой системы, а при вызове диспетчером ввода-вывода в процессе монтирования файловой системы для нового тома он загружает драйвер соответствующей файловой системы, если такой драйвер еще не загружен. После этого File System Recognizer перенаправляет IRP монтирования драйверу и позволяет ему захватить том во владение.

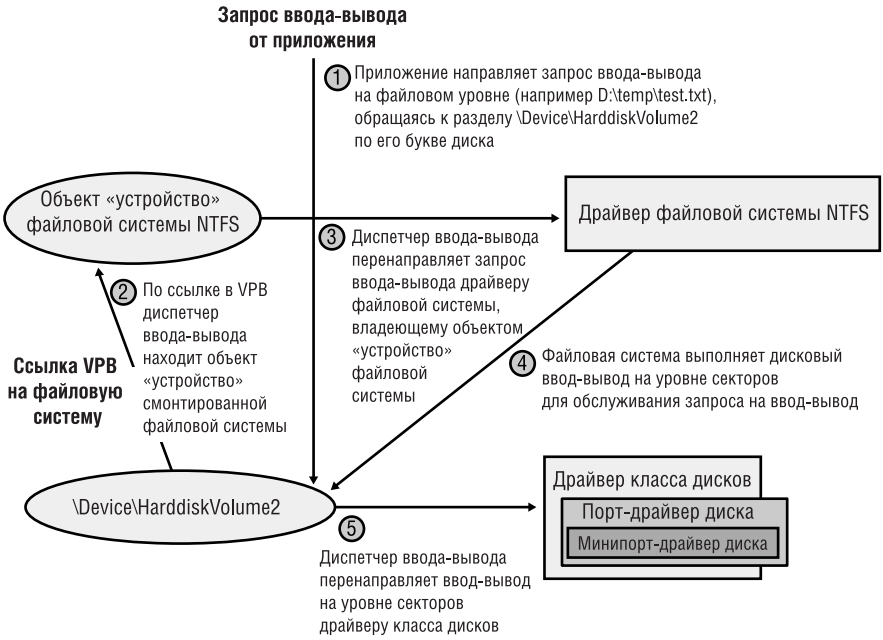


Рис. 10-15. *Поток ввода-вывода на смонтированном томе*

Кроме загрузочного тома, чей драйвер монтируется при инициализации ядра, драйверы файловых систем монтируют большинство томов в момент запуска Chkdsk для проверки целостности файловой системы на этапе загрузки системы. Загрузочная версия Chkdsk является встроенным приложением (в отличие от Windows-приложений) и называется Autochk.exe (\Windows \System32\Autochk.exe). Диспетчер сеансов (\Windows\System32 \Smss.exe) запускает ее, поскольку она указана в параметре HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\BootExecute. Chkdsk перебирает все назначенные буквы диска, чтобы выяснить, требует ли соответствующий том проверки целостности.

Один и тот же сменный носитель может монтироваться более чем раз. Драйверы файловых систем Windows реагируют на смену носителей и зап-

рашивают идентификатор тома. Если они обнаруживают, что идентификатор тома сменился, драйверы демонтируют диск и монтируют его заново.

Операции ввода-вывода на томах

Драйверы файловых систем управляют хранящимися в томах данными, но требуют поддержки диспетчера томов для взаимодействия с драйверами устройств внешней памяти при передаче данных. Драйверы файловых систем получают ссылки на объекты томов в процессе монтирования и посылают через них запросы диспетчеру томов. Приложения — если им нужно напрямую обращаться к данным тома — тоже могут посылать запросы диспетчеру томов, обходя драйвер файловой системы. К числу таких приложений относятся, например, программы восстановления удаленных файлов и утилита DiskProbe из ресурсов Windows.

Когда драйвер файловой системы или приложение посылает объекту «устройство», представляющему том, запрос ввода-вывода, диспетчер ввода-вывода перенаправляет этот запрос (поступающий в виде IRP) диспетчеру томов, создавшему целевой объект «устройство». Таким образом, если приложению нужно считать загрузочный сектор, например, второго простого тома в системе, оно открывает объект \Device\HarddiskVolume2 и посылает ему запрос на чтение 512 байтов по нулевому смещению на устройстве. Диспетчер ввода-вывода передает запрос приложения в виде IRP диспетчеру томов, владеющему данным объектом «устройство», и уведомляет его, что IRP адресован устройству HarddiskVolume2.

Поскольку том логически представляет непрерывную область одного или более физических дисков, диспетчер томов должен преобразовывать смещения, относительные началу тома, в смещения, относительные началу диска. Если том 2 состоит из одного раздела, который начинается с 4096-го сектора диска, то, прежде чем передать запрос драйверу класса дисков, диспетчер томов соответственно корректирует параметры IRP. Для выполнения ввода-вывода на физическом диске и чтения запрошенных данных в буфер приложения, указанный в IRP, драйвер класса дисков использует минипорт-драйвер.

Роль диспетчера томов в обработке запросов к составным томам помогут прояснить следующие примеры. Если чередующийся том состоит из двух разделов (1 и 2), представленных объектом \Device\HarddiskDmVolumes\PhysicalDmVolumes\BlockVolume3 (рис. 10-16), и администратор назначил чередующейся области букву диска D:, то диспетчер ввода-вывода определяет ссылку \Global??\D:, указывающую на \Device\HarddiskDmVolumes*ComputerName*Dg0\Volume3, где *ComputerName* — имя компьютера. Вспомните, что эта ссылка также является символьной и указывает на соответствующий объект тома в каталоге PhysicalDmVolumes (в данном случае — на BlockVolume3). Объект «устройство», принадлежащий DMIO, перехватывает дисковый ввод-вывод файловой системы на \Device\HarddiskDmVolumes\PhysicalDmVolumes\BlockVolume3, и драйвер DMIO корректирует параметры запроса перед тем, как передать его драйверу класса дисков. В результате изменений, внесенных DMIO, запрос настраивается так, чтобы он ссылался на нужное

смещение, относительное началу целевой чередующейся области раздела 1 или 2. Если ввод-вывод затрагивает оба раздела тома, DMIO должен выдать два дополнительных запроса ввода-вывода — по одному к каждому диску.

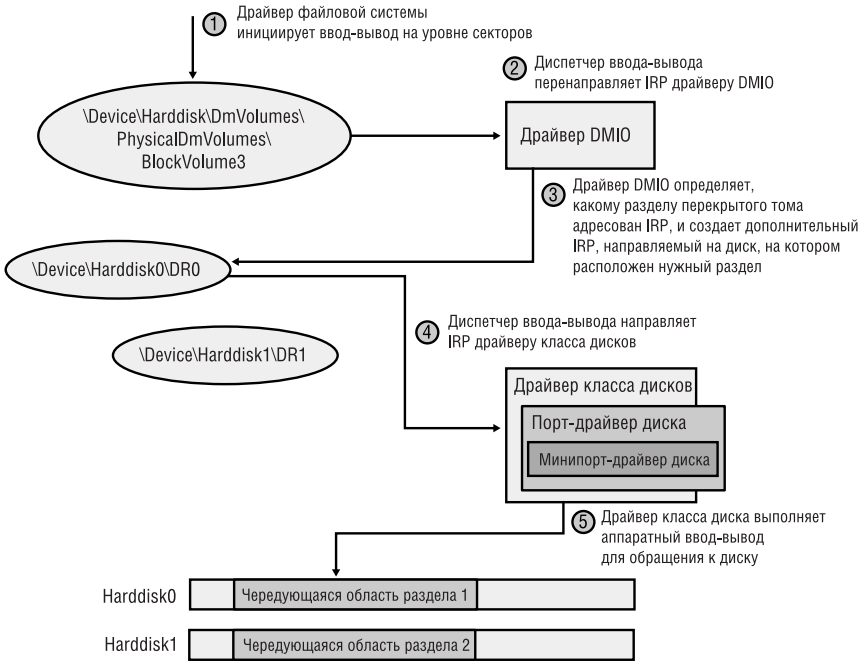


Рис. 10-16. *Операции ввода-вывода DMIO*

В случае записи на зеркальный том DMIO делит каждый запрос так, что операция записи выполняется над каждой половиной зеркального тома. А при запросе на чтение с зеркального тома DMIO использует одну из половин зеркального тома и обращается к другой половине, только если первая попытка чтения заканчивается неудачно.

Служба виртуального диска

Компания, которая выпускает продукты, имеющие отношение к внешней памяти, например RAID-адаптеры, жесткие диски или массивы накопителей, вынуждена реализовать собственные приложения для установки этих устройств и управления ими. Применение разных управляющих приложений для разных устройств внешней памяти имеет очевидные недостатки с точки зрения системного администрирования, например приходится изучать множество интерфейсов и нельзя использовать стандартные Windows-утилиты для управления сторонними устройствами внешней памяти.

В Windows Server 2003 введена служба виртуального диска (Virtual Disk Service, VDS) (\Windows\System32\Vds.exe), которая предоставляет системным администраторам унифицированный высокоуровневый интерфейс

внешней памяти; благодаря этому устройствами внешней памяти от разных поставщиков можно управлять через одни и те же пользовательские интерфейсы (UI). Схема VDS представлена на рис. 10-17. VDS экспортирует API, основанный на COM и позволяющий приложениям и сценариям создавать и форматировать диски, а также управлять аппаратными RAID-адаптерами. Скажем, утилита может задействовать VDS API для запроса списка физических дисков, сопоставленных с номером логического блока RAID (logical unit number, LUN). Windows-средства управления дисками, включая оснастку Disk Management консоли MMC, Diskpart и Diskraid (поставляется с Windows Server 2003 Deployment Kit), тоже используют VDS API.

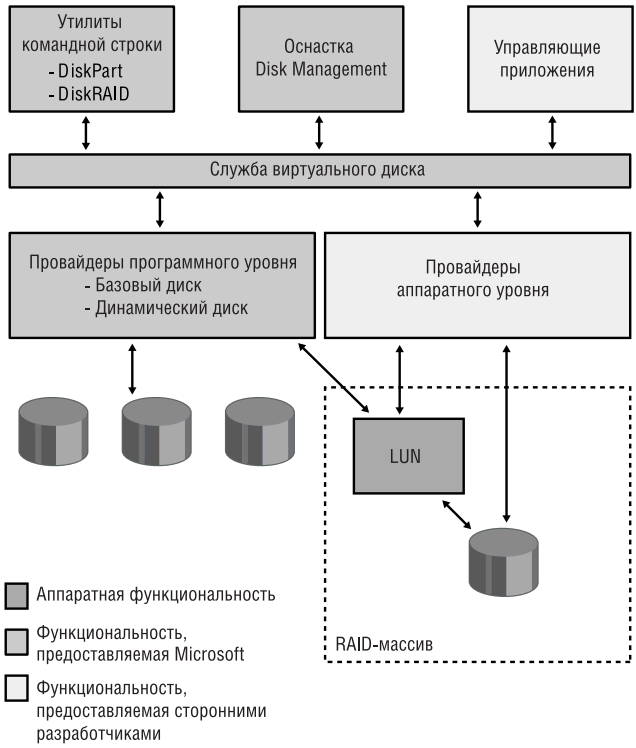


Рис. 10-17. *Архитектура VDS*

VDS предоставляет два интерфейса: один — для провайдеров программного уровня, другой — для провайдеров аппаратного уровня.

- *Провайдеры программного уровня* (software providers) реализуют интерфейсы к таким высокоуровневым абстракциям устройств внешней памяти, как диски, разделы дисков и тома. Примеры операций, поддерживаемых этими интерфейсами, — расширение и удаление томов, включение и отключение зеркалирования, форматирование томов и присвоение им букв дисков. VDS ищет зарегистрированные программные провайдеры в HKLM\System\CurrentControlSet\Services\Vds\SoftwareProviders. Windows

Server 2003 включает VDS Dynamic Disk Provider (\Windows\System32\Vdsdyndr.dll), применяемый в качестве интерфейса для динамических дисков, и VDS Basic Provider (\Windows\System32\Vdsbas.dll), используемый в качестве интерфейса для базовых дисков.

- *Провайдеры аппаратного уровня* (hardware providers) реализуются изготовителями оборудования в виде DLL, которые регистрируются в разделе реестра HKLM\System\CurrentControlSet\Services\Vds\HardwareProviders и которые транслируют аппаратно-независимые VDS-команды в команды, специфичные для конкретного оборудования. Провайдер аппаратного уровня позволяет управлять подсистемой внешней памяти, например аппаратным RAID-массивом или платами адаптеров/контроллеров, и поддерживает такие операции, как создание, расширение, удаление, маскирование и отмена маскирования LUN.

Когда приложение инициирует соединение с VDS API и служба VDS еще не запущена, процесс Svchost — хост службы RPC запускает процесс загрузчика VDS (\Windows\System32\Vdsldr.exe), а тот — процесс службы VDS, после чего завершается. После закрытия последнего соединения с VDS API завершается и процесс службы VDS.

Служба теневого копирования тома

Одно из ограничений многих утилит резервного копирования связано с открытыми файлами. Если приложение открывает какой-нибудь файл для монопольного доступа, утилита резервного копирования не может получить доступа к содержимому этого файла. Но даже если подобная утилита способна обращаться к уже открытому файлу, нет никаких гарантий, что его резервная копия не окажется в рассогласованном состоянии. Допустим, приложение обновляет начальную часть файла, а потом что-то пишет в его конце. Утилита резервного копирования, которая сохраняет файл в ходе этих операций, может записать такой образ файла, который отражает еще не модифицированную начальную часть файла и уже измененную концевую часть. При последующем восстановлении этого файла приложение сочтет, что файл поврежден, поскольку оно допускает ситуации, в которых начальная часть уже изменена, а концевая — еще нет, но только не наоборот. Именно поэтому большинство утилит резервного копирования пропускает открытые файлы.

В связи с этим в Windows XP появилась служба теневого копирования томов (Volume Shadow Copy Service) (\Windows\System32\Vssvc.exe), которая позволяет встроенной утилите резервного копирования записывать согласованные представления всех файлов, в том числе открытых. Эта служба выступает в роли командного центра расширяемого механизма резервного копирования, давая возможность независимым поставщикам программного обеспечения (independent software vendors, ISV) подключать свои провайдеры и модули записи («writers»). Модуль записи — это программный компонент, позволяющий приложениям с поддержкой теневого копирования томов принимать уведомления о замораживании и размораживании операций записи,

чтобы они могли создавать внутренне согласованные резервные копии своих файлов данных. А провайдеры позволяют ISV интегрировать уникальные схемы работы с внешней памятью со службой теневого копирования томов. Например, приложение, использующее устройства внешней памяти с зеркалированием, могло бы определять теневую копию как замороженную половину зеркалированного тома. Взаимосвязи между службой теневого копирования томов, модулями записи и провайдерами показаны на рис. 10-18.

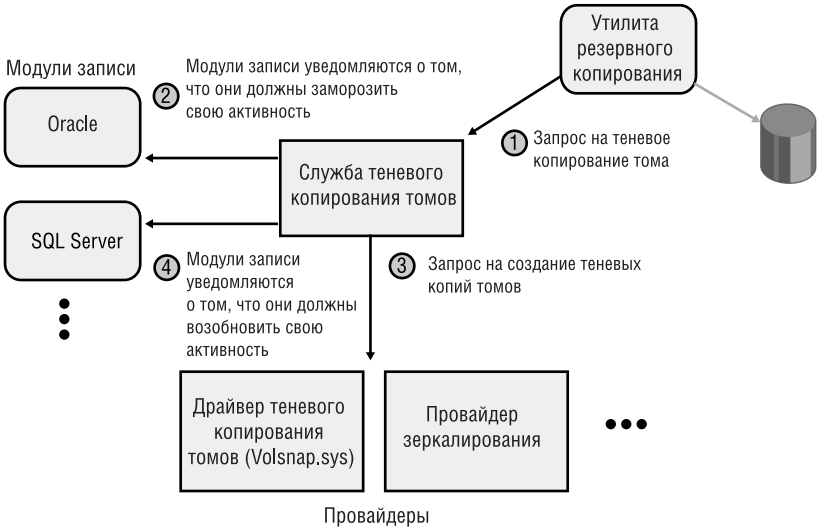


Рис. 10-18. *Служба теневого копирования томов, модули записи и провайдеры*

Microsoft Shadow Copy Provider (\Windows\System32\Drivers\Volsnap.sys) — это провайдер, поставляемый с Windows для поддержки программных снимков томов. Он представляет собой драйвер фильтра внешней памяти, размещаемый между драйверами файловых систем и драйверами томов (они оперируют с наборами секторов на жестком диске, представляющими логические диски), и поэтому видит любые запросы на ввод-вывод, адресованные дисковому тому. Утилита резервного копирования, приступая к копированию, указывает драйверу Microsoft Shadow Copy Provider создать теневые копии всех томов, на которых содержатся копируемые файлы и каталоги. Драйвер замораживает все операции ввода-вывода на этих томах и для каждого из них создает теневую копию. Если, например, том в пространстве имен диспетчера объектов имеет имя \Device\HarddiskVolume0, то теневой том получает имя в виде \Device\HarddiskVolumeShadowCopy*N*, где *N* — уникальный идентификатор.

ЭКСПЕРИМЕНТ: просмотр объектов «устройство», принадлежащих драйверу Microsoft Shadow Copy Provider

Чтобы просмотреть такие объекты, связанные с каждым томом, в Windows XP или Windows Server 2003, используйте отладчик ядра. В любой системе есть хотя бы один том, и следующая команда выводит информацию об объекте «устройство» для первого тома в системе:

```
0: kd> !devobj \device\harddiskvolume1
Device object (86b9daf8) is for:
 HarddiskVolume1 \Driver\Ftdisk DriverObject 86b6af38
Current Irp 00000000 RefCount 2 Type 00000007 Flags 00001050
Vpb 86b90008 Dacl e1000384 DevExt 86b9dbb0 DevObjExt 86b9dc98
Dope 86ba33e8 DevNode 86b71320
ExtensionFlags (0xa8000000)  DOE_RAW_FDO, DOE_DESIGNATED_FDO
                             Unknown flags 0x08000000
AttachedDevice (Upper) 86b687f8 \Driver\VolSnap
Device queue is not busy.
```

Поле *AttachedDevice* в выводе команды *!devobj* сообщает адрес объекта «устройство» и имя владеющего им драйвера, который подключен к этому объекту (фильтрует его). Каждый объект «устройства» для тома должен принадлежать драйверу Volsnap, как в показанном примере.

Вместо того чтобы открывать копируемые файлы на исходном томе, утилита резервного копирования открывает их на теневом. Последний отражает представление тома, привязанное к определенной временной точке (point-in-time view of a volume). Поэтому, когда драйвер теневого копирования томов обнаруживает попытку записи на исходный том, он считывает копию подлежащих перезаписи секторов в раздел памяти, поддерживаемый страничным файлом (paging file-backed memory section) и сопоставленный с соответствующим теневым томом. Обращения для чтения к модифицируемым секторам теневого тома драйвер обслуживает через упомянутый выше раздел памяти, а обращения для чтения к немодифицированным секторам — считыванием данных с исходного тома. Поскольку утилита резервного копирования не сохраняет страничный файл и системный каталог \System Volume Information (вместе со всеми подкаталогами и файлами), драйвер снимков, используя API-функции дефрагментации, определяет местонахождение этих файлов и каталогов и не регистрирует вносимые в них изменения. Опираясь на данный механизм, утилита резервного копирования в Windows XP и Windows Server 2003 решает все проблемы копирования, связанные с открытыми файлами.

Рис. 10-19 иллюстрирует, как ведут себя приложение, обращающееся к тому, и утилита резервного копирования, обращающаяся к теневой копии этого тома. Когда приложение пишет в сектор по истечении времени снятия снимка, драйвер Volsnap создает резервную копию, как на иллюстрации, где он копирует секторы a, b и c для тома C:. Аналогично, когда приложение

считывает сектор c, Volsnap направляет операцию чтения тому C:, а когда утилита резервного копирования считывает тот же сектор, Volsnap получает содержимое этого сектора из снимка. Если операция чтения требует обращения к немодифицированному сектору, например к d, то Volsnap направляет ее исходному тому.

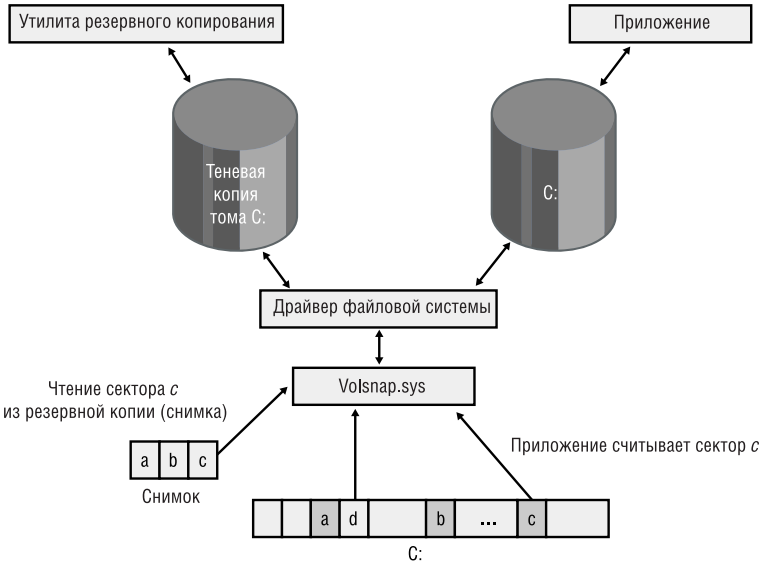


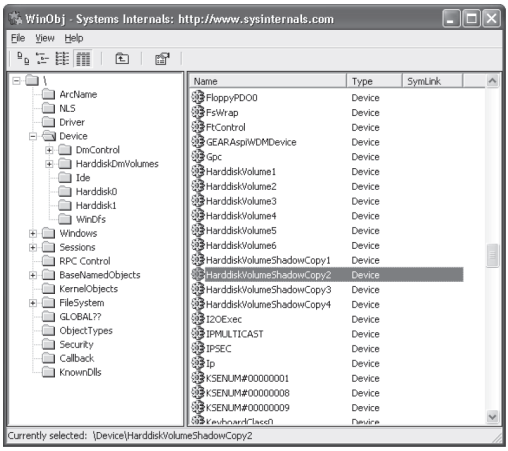
Рис. 10-19. *Так работает Volsnap*

ЭКСПЕРИМЕНТ: просмотр объектов «устройство» теневых томов

Вы можете убедиться в наличии таких объектов в пространстве имен диспетчера объектов, запустив Windows-утилиту резервного копирования [в меню Start (Пуск) откройте Accessories (Стандартные) и System Tools (Служебные)] и выбрав достаточный объем данных для резервного копирования, чтобы успеть запустить Winobj и просмотреть объекты в подкаталоге \Device.

Драйверы файловых систем должны корректно обрабатывать два запроса на управление вводом-выводом (IOCTL), связанные с теневым копированием томов: IOCTL_VOLSNAP_FLUSH_AND_HOLD_WRITES и IOCTL_VOLSNAP_RELEASE_WRITES. Смысл этих запросов не требует объяснений — он понятен из их имен. API копирования теневых томов позволяет посылать IOCTL-запросы логическим дискам, для которых создаются снимки, с тем чтобы все операции записи, инициированные перед получением снимка, успели завершиться до создания теневой копии и чтобы файловые данные, записываемые из теневой копии, были согласованы по времени.

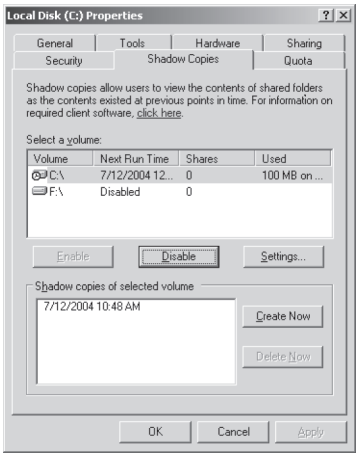
см. след. стр.



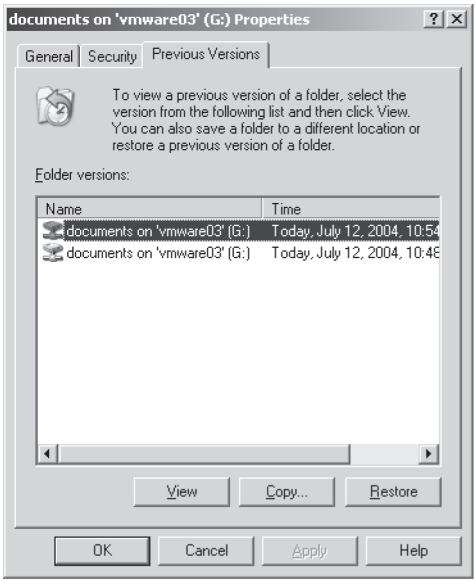
Теневые копии для общих папок

Поддержка теневого копирования томов позволяет Windows Server 2003 предоставлять конечным пользователям доступ к резервным версиям томов для восстановления старых версий файлов и папок, которые могли быть случайно удалены или изменены. Эта функция облегчает жизнь системным администраторам, которые в ином случае должны были бы загружать резервные данные и обращаться к предыдущим их версиям в интересах конечных пользователей.

В окне свойств для тома в Windows Server 2003 есть вкладка Shadow Copies (Теневые копии), на которой администратор может разрешить создание снимков томов по расписанию, как показано на следующей иллюстрации. Администраторы также могут ограничить пространство, выделяемое под снимки, чтобы система автоматически удаляла самые старые снимки.



В клиентских системах, где нужна функциональность Shadow Copies for Shared Folders, следует установить расширение Explorer — Previous Versions Client — которое поставляется с Windows Server 2003 в каталоге \Windows\System32\Clients\Twclient и которое также можно скачать с сайта Microsoft (это расширение включено в Windows XP Service Pack 2 и выше). Когда клиентская Windows-система с установленным расширением подключается к общей папке на томе, для которого имеются снимки, в окне свойств папок и файлов, находящихся в этой общей папке, появляется вкладка Previous Versions. На этой вкладке перечисляются снимки, имеющиеся на сервере, и пользователь может просмотреть соответствующие версии файла или папки.



Резюме

В этой главе мы рассмотрели организацию, компоненты и принципы управления внешней дисковой памятью в Windows. В следующей главе мы обсудим диспетчер кэша — компонент исполнительной системы, неразрывно связанный с драйверами файловых систем.

ГЛАВА 11

Диспетчер кэша

Диспетчер кэша (cache manager) — это набор функций режима ядра и системных потоков, во взаимодействии с диспетчером памяти обеспечивающих кэширование данных для всех драйверов файловых систем Windows (как локальных, так и сетевых). В этой главе мы поясним, как работает диспетчер кэша, что представляют собой его внутренние структуры данных и функции, как определяется размер кэшей при инициализации системы, как он взаимодействует с другими компонентами операционной системы и каким образом можно наблюдать за его активностью с помощью счетчиков производительности. Мы также рассмотрим пять флагов Windows-функции *CreateFile*, влияющих на кэширование файлов.

ПРИМЕЧАНИЕ В этой главе описываются лишь те внутренние функции диспетчера кэша, которые нужны для объяснения принципов его работы. Программные интерфейсы диспетчера кэша документированы в Windows Installable File System (IFS) Kit. Подробнее об IFS Kit см. *http://www.microsoft.com/whdc/devtools/ifskit*.

Основные возможности диспетчера кэша

Диспетчер кэша:

- поддерживает все файловые системы Windows (как локальные, так и сетевые), исключая необходимость реализации в каждой файловой системе собственного кода управления кэшем;
- с помощью диспетчера памяти контролирует, какие части и каких файлов находятся в физической памяти (обеспечивая компромисс между потребностями в физической памяти пользовательских процессов и операционной системы);
- в отличие от большинства других систем кэширования, которые кэшируют данные на основе логических блоков (смещений внутри дисковых томов), кэширует данные на основе виртуальных блоков (смещений внутри файлов), что позволяет реализовать алгоритм интеллектуального опережающего чтения и обеспечить высокоскоростной доступ к кэшу без участия драйверов файловых систем (этот метод кэширования называется быстрым вводом-выводом);

- распознает параметры, передаваемые приложениями при открытии файлов (например, прямой или последовательный доступ, временный файл или постоянный и т. д.);
- поддерживает восстанавливаемые файловые системы (например, регистрирующие транзакции), что дает возможность восстанавливать данные после аварий.

Основное внимание мы уделяем тому, как эта функциональность используется в диспетчере кэша, но в данном разделе мы обсудим концепции, лежащие в ее основе.

Единый централизованный системный кэш

В некоторых операционных системах данные кэшируются каждой файловой системой индивидуально. Это приводит к дублированию кода, отвечающего за кэширование и управление памятью, или к ограничению видов данных, которые можно кэшировать. В противоположность этому подходу Windows предлагает централизованный механизм кэширования всех данных, хранящихся во внешней памяти — на локальных жестких и гибких дисках, сетевых файл-серверах или CD-ROM. Кэшировать можно любые данные — как пользовательские (содержимое файлов при операциях чтения или записи), так и метаданные файловой системы (например, заголовки каталогов и файлов). Как вы еще узнаете из этой главы, метод обращения к кэшу, применяемый Windows, определяется типом кэшируемых данных.

Диспетчер памяти

Одно весьма необычное свойство диспетчера кэша заключается в том, что он никогда не знает, какая часть кэшируемых данных действительно находится в физической памяти. Вероятно, это звучит несколько странно, поскольку кэш предназначен для ускорения ввода-вывода за счет хранения в физической памяти подмножества данных, к которым часто обращаются приложения и система. Все дело в том, что диспетчер кэша обращается к данным, проецируя представления файлов на виртуальные адресные пространства с помощью стандартных объектов «раздел» (в терминах Windows API — объектов «проекция файла»; см. главу 7). По мере доступа к адресам проецируемых представлений файлов диспетчер памяти подгружает нерезидентные блоки в физическую память. А при необходимости диспетчер памяти может выгружать данные из кэша обратно в файлы, проецируемые на кэш.

Используя кэширование на основе проецирования файлов на виртуальное адресное пространство, диспетчер кэша избегает генерации пакетов запроса ввода-вывода (IRP) при обращении к данным кэшируемых файлов. Вместо этого он просто копирует данные по виртуальным адресам, по которым проецируются кэшируемые данные, а диспетчер памяти при необходимости подгружает данные в память (или выгружает их из нее). Этот процесс позволяет диспетчеру памяти подбирать глобальный баланс между объемом памяти, выделенной системному кэшу, и объемом памяти, нужной пользовательским

процессам. (Диспетчер кэша также инициирует ввод-вывод, например отложенную запись, но сама запись страниц осуществляется диспетчером памяти.) Как вы узнаете из следующего раздела, такая архитектура дает возможность процессам, открывающим кэшируемые файлы, видеть те же данные, что и процессам, проецирующим эти файлы на свои адресные пространства.

Когерентность кэша

Одна из важных функций диспетчера кэша — гарантировать любому процессу, обращающемуся к кэшируемым данным, получение самой последней версии этих данных. Ситуация, при которой один процесс открывает файл (и, следовательно, делает его кэшируемым), тогда как другой напрямую проецирует этот файл на свое адресное пространство (через Windows-функцию *MapViewOfFile*), может обернуться проблемой. Эта потенциальная проблема не возникает в Windows, поскольку и диспетчер кэша, и приложения, проецирующие файлы на свои адресные пространства, используют одни и те же сервисы подсистемы управления памятью. Так как диспетчер памяти гарантирует, что у него имеется только одно представление каждого уникального проецируемого файла (независимо от количества объектов «раздел», или проекций файла), он проецирует все представления файла (даже если они перекрываются) на единственный набор страниц физической памяти, как показано на рис. 11-1.

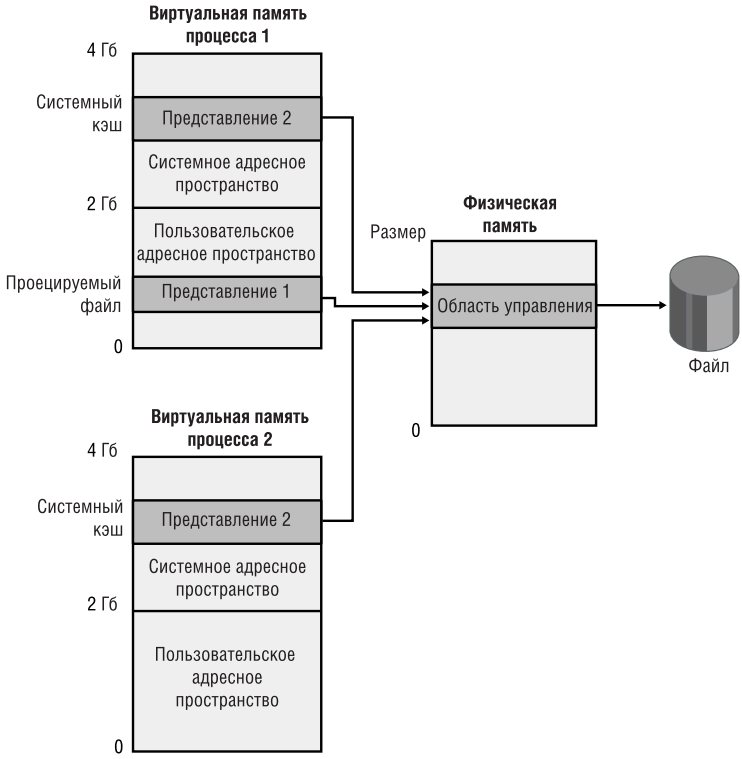


Рис. 11-1. *Схема когерентного кэширования*

Поэтому, если, например, на пользовательское адресное пространство процесса 1 проецируется представление 1 файла и процесс 2 обращается к тому же представлению через системный кэш, то процесс 2 будет видеть любые изменения в этом представлении по мере их внесения процессом 1, а не после сброса измененных данных из кэша на диск. Диспетчер памяти сбрасывает не все страницы, проецируемые на пользовательские пространства, а лишь те, о которых он знает, что они изменены (установлен бит изменения). Поэтому любой процесс, обращающийся к файлу, всегда видит его самую последнюю версию, даже если одни процессы открыли этот файл через подсистему ввода-вывода, а другие проецируют его на свои адресные пространства с помощью соответствующих Windows-функций.

ПРИМЕЧАНИЕ В силу некоторых причин сетевым редиректорам труднее поддерживать когерентность кэша, чем локальным файловым системам, и они должны реализовать дополнительные операции сброса и очистки кэша. На эту тему см. главу 12.

Кэширование виртуальных блоков

Диспетчеры кэша многих операционных систем (включая Novell NetWare, OpenVMS и ранние версии UNIX) кэшируют данные на основе *логических блоков*. В этом случае диспетчер кэша отслеживает, какие блоки дискового раздела находятся в кэше. Диспетчер кэша Windows, напротив, использует *кэширование виртуальных блоков*. Этот метод заключается в том, что диспетчер кэша отслеживает, какие части и каких файлов находятся в кэше. Диспетчер кэша реализует этот метод за счет проецирования их 256-килобайтных представлений на системную часть виртуальных адресных пространств с помощью специальных процедур диспетчера памяти. Основные преимущества такого подхода описываются ниже.

- Появляется возможность реализации интеллектуального опережающего чтения (intellegent read-ahead), так как диспетчер кэша следит за тем, части каких файлов находятся в кэше, и это позволяет ему предсказывать, к какой следующей порции данных обратится вызывающая программа.
- Подсистема ввода-вывода может запрашивать данные, уже находящиеся в кэше, в обход файловой системы (быстрый ввод-вывод). Поскольку диспетчеру кэша известно, какие части и каких файлов находятся в кэше, он может вернуть адрес кэшируемых данных для выполнения запроса на ввод-вывод без обращения к файловой системе.

Подробнее об интеллектуальном опережающем чтении и быстром вводе-выводе мы расскажем чуть позже.

Кэширование потоков данных

В диспетчер кэша заложена поддержка не только кэширования файлов, но и *кэширования потоков данных* (stream caching) — последовательности байтов в файле. В файлах таких файловых систем, как NTFS, может быть более одно-

го потока данных. Диспетчер кэша поддерживает эти файловые системы за счет независимого кэширования каждого потока. NTFS способна использовать эту функциональность (см. главу 12). И хотя о диспетчере кэша можно сказать, что он кэширует файлы, фактически он кэширует именно потоки данных (в любом файле есть минимум один поток данных), идентифицируемые по имени файла и, если в нем более одного потока, по имени потока.

Поддержка восстанавливаемых файловых систем

Восстанавливаемые файловые системы вроде NTFS способны реконструировать структуру дискового тома после аварии системы. Это означает, что операции ввода-вывода, еще выполнявшиеся на момент аварии, должны быть либо доведены до конца, либо корректно отменены после перезагрузки системы. Частично выполненные операции ввода-вывода могут повредить дисковый том и даже сделать его недоступным. Во избежание такой проблемы восстанавливаемая файловая система ведет файл журнала, в котором регистрирует каждое предполагаемое обновление структуры файловой системы (метаданные файловой системы) — еще до того, как оно будет выполнено. Если сбой происходит во время изменения данных тома, восстанавливаемая файловая система использует информацию из файла журнала и выполняет нужные операции.

> **ПРИМЕЧАНИЕ** Термин *метаданные* относится только к изменениям в структуре файловой системы в результате создания, переименования и удаления файлов и каталогов.

Чтобы обеспечить успешное восстановление тома, каждый элемент (запись) файла журнала, документирующий изменения в данных тома, должен быть записан на диск до самого обновления данных. Поскольку запись на диск кэшируется, файловая система должна взаимодействовать с диспетчером кэша, чтобы гарантировать выполнение следующей последовательности операций.

1. Файловая система заносит в файл журнала запись, документирующую изменение данных тома, которое она собирается выполнить.
2. Файловая система вызывает диспетчер кэша для сброса на диск записи файла журнала.
3. Файловая система записывает в кэш обновленные данные тома, т. е. модифицирует свои кэшируемые метаданные.
4. Диспетчер кэша сбрасывает модифицированные метаданные на диск, обновляя структуру тома. (На самом деле записи файла журнала, как и модифицированные метаданные, сбрасываются на диск пакетами.)

Записывая данные в кэш, файловая система предоставляет *номер логической последовательности* (logical sequence number, LSN), который идентифицирует запись файла журнала, соответствующую обновлению кэша. Диспетчер кэша отслеживает эти номера, регистрируя наименьший и наибольший LSN (представляющие первую и последнюю записи файла журнала); такие номера сопоставляются с каждой страницей кэша. Кроме того, потоки данных, защищенные записями журнала транзакций, помечаются NTFS как «не записывае-

мые», чтобы подсистема записи спроецированных страниц не сбросила эти страницы на диск до того, как туда будут сброшены соответствующие записи файла журнала. (Обнаружив помеченную таким образом страницу, подсистема записи спроецированных страниц перемещает ее в специальный список страниц, которые диспетчер кэша сбрасывает на диск в подходящий момент.)

Когда диспетчер кэша готов сбросить на диск группу измененных страниц, он определяет наибольший LSN, сопоставленный со сбрасываемыми на диск страницами, и сообщает его файловой системе. Далее файловая система может ответно вызвать диспетчер кэша и заставить его сбросить данные файла журнала вплоть до точки, представленной указанным LSN. Сбросив данные файла журнала вплоть до указанного LSN, диспетчер кэша сбрасывает на диск и соответствующие обновления в структуре тома, что гарантирует регистрацию предстоящих операций до их выполнения. Таким образом и достигается возможность восстановления дискового тома после аварии системы.

Управление виртуальной памятью кэша

Поскольку диспетчер кэша Windows кэширует файлы на основе виртуальных блоков, ему передается регион в системной части виртуальных адресных пространств (а не область физической памяти). Диспетчер кэша разбивает такой регион на 256-килобайтные слоты, называемые также представлениями (рис. 11-2). (Подробнее о структуре системного пространства см. главу 7.)

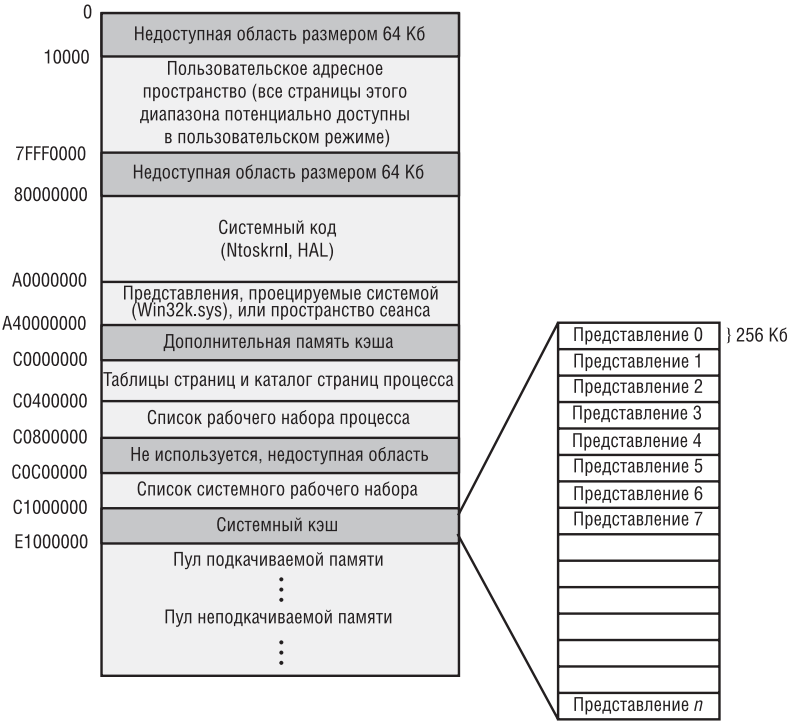


Рис. 11-2. *Адресное пространство системного кэша*

При первой операции ввода-вывода (чтения или записи) над файлом диспетчер кэша проецирует на свободный слот адресного пространства системного кэша 256-килобайтное представление области файла, выровненной по границе 256 Кб и содержащей запрошенные данные. Например, если из файла считывается 10 байтов по смещению 300 000 байтов от его начала, то проецируемое представление будет начинаться со смещения 262 144 (вторая область файла, выровненная по границе 256 Кб) и займет 256 Кб.

Диспетчер кэша проецирует представления файлов на слоты адресного пространства кэша по принципу карусели: первое запрошенное представление — на первый 256-килобайтный слот, второе — на второй и т. д. (рис. 11-3). В этом примере первым был спроецирован файл B, вторым — A, третьим — C, поэтому проецируемая часть файла B занимает первый слот кэша. Заметьте, что спроецирована лишь первая 256-килобайтная часть файла B, так как обращение было лишь к части файла и так как файл C, размер которого составляет всего 100 Кб, требует выделения своего 256-килобайтного слота кэша.

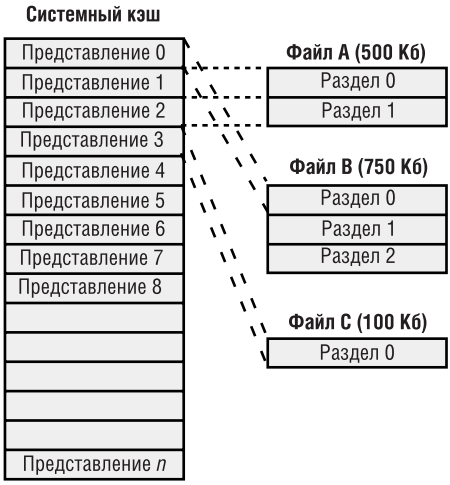


Рис. 11-3. *Файлы различного размера, спроецированные в системный кэш*

Диспетчер кэша гарантирует, что представление проецируется на то время, пока оно активно (хотя представления могут оставаться спроецированными после того, как становятся неактивными). Однако представление помечается как активное, только когда выполняется операция чтения или записи над соответствующим файлом. Если процесс, открывающий файл вызовом *CreateFile*, не указывает флаг FILE_FLAG_RANDOM_ACCESS, диспетчер кэша прекращает проецировать неактивные представления этого файла при проецировании его новых представлений. Страницы отключенных проекций посылаются в список простаивающих или модифицированных страниц (в зависимости от того, были ли они изменены); при этом диспетчер кэша, используя специальный интерфейс диспетчера памяти, может указать, в каком месте списка следует разместить эти страницы — в конце или в начале.

Страницы, соответствующие представлениям файлов, открытых с флагом FILE_FLAG_SEQUENTIAL_SCAN, перемещаются в начало списков, а все остальные — в конец. Такая схема способствует повторному использованию страниц, которые принадлежат файлам, открытым для последовательного чтения, и заставляет использовать малые объемы физической памяти при копировании больших файлов.

Если диспетчеру кэша требуется спроецировать представление файла, а свободных слотов в кэше нет, он отключает неактивное представление, спроецированное последним, и использует освободившийся слот. В отсутствие таких представлений возвращается ошибка ввода-вывода с сообщением о том, что системных ресурсов для выполнения данной операции недостаточно. Эта ситуация крайне маловероятна, так как возникает только при одновременном доступе к тысячам файлов.

Размер кэша

В следующих разделах мы объясним, как Windows вычисляет размер системного кэша. Как и в большинстве других вычислений, связанных с управлением памятью, размер системного кэша определяется несколькими факторами, в том числе объемом памяти и конкретным выпуском Windows.

LargeSystemCache

Как вы увидите в дальнейшем, параметр LargeSystemCache в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management влияет как на виртуальный размер кэша, так и на физический. По умолчанию в Windows 2000 Professional и Windows XP это значение равно 0, а в системах Windows Server — 1. В Windows 2000 Server данное значение можно регулировать через GUI, изменяя свойства службы файлового сервера; для этого надо открыть окно свойств сетевого соединения и выбрать File And Printer Sharing For Microsoft Networks (Служба доступа к файлам и принтерам сетей Microsoft). Эта служба имеется и в Windows 2000 Professional, но там ее параметры настраивать нельзя. На рис. 11-4 показано диалоговое окно, через которое в Windows 2000 Server можно изменить объем памяти, выделяемой для локальных и сетевых приложений сетевой службой сервера.

В Windows 2000 Server с установленными Terminal Services (Службы терминала) переключатель Maximize Data Throughput For File Sharing (Макс. пропускная способность доступа к общим файлам), показанный на рис. 11-4, активен по умолчанию, т. е. параметр LargeSystemCache равен 1. При выборе любого другого переключателя параметр LargeSystemCache становится равным 0. Каждый из переключателей диалогового окна File And Printer Sharing For Microsoft Networks Properties влияет не только на поведение системного кэша, но и на службу файлового сервера.

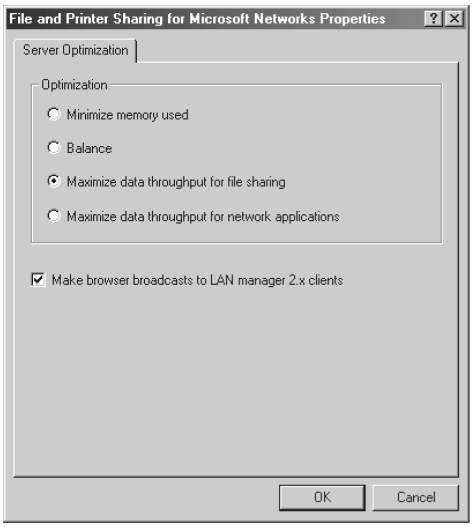


Рис. 11-4. *Диалоговое окно File and Printer Sharing for Microsoft Networks Properties, позволяющее изменять свойства сетевой службы сервера*

В Windows XP и Windows Server 2003 модифицировать параметр LargeSystemCache можно через диалоговое окно Performance Options (Параметры быстродействия), которое открывается щелчком кнопки Settings (Параметры) в разделе Performance (Быстродействие) на вкладке Advanced (Дополнительно) апплета System (Система) из Control Panel (Панель управления). В этом диалоговом окне перейдите на очередную вкладку Advanced (Дополнительно). Если в разделе Memory Usage (Использование памяти) вы выбираете System Cache (системного кэша), параметру LargeSystemCache присваивается значение 1, а если вы выбираете Programs (программ) — 0 (рис. 11-5).

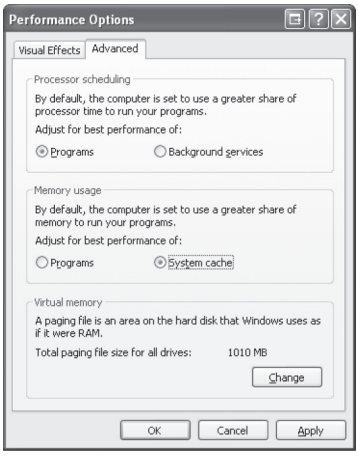


Рис. 11-5. *Настройка LargeSystemCache в Windows XP и Windows Server 2003*

Виртуальный размер кэша

Виртуальный размер системного кэша является функцией объема установленной физической памяти. По умолчанию это значение равно 64 Мб. Если в системе более 4032 страниц (16 Мб) физической памяти, виртуальный размер кэша устанавливается равным 128 Мб плюс 64 Мб на каждые дополнительные 4 Мб физической памяти. Используя этот алгоритм, можно подсчитать виртуальный размер системного кэша на компьютере, например, с 64 Мб физической памяти:

```
128 Мб + (64 Мб – 16 Мб) / 4 Мб * 64 Мб = 896 Мб
```

Минимальный и максимальный виртуальные размеры системного кэша на разных платформах, а также его стартовый и конечный адреса показаны в таблице 11-1. Если на платформе x86 рассчитанный системой виртуальный размер кэша превышает 512 Мб, он ограничивается 512 Мб; однако при параметре LargeSystemCache, равном 1, в той же ситуации кэшу назначается до 960 Мб виртуальной памяти из дополнительного диапазона адресов, называемого *дополнительной памятью кэша* (cache extra memory). Основное преимущество выделения под кэш большего объема виртуальной памяти заключается в том, что это позволяет уменьшить число операций проецирования и отмены проецирования представлений при обращении к разным файлам и разным представлениям файлов.

Таблица 11-1. *Размер и местонахождение системного кэша данных*

Платформа	Диапазон адресов	Минимальный/ максимальный виртуальный размер
x86, 2-гигабайтное системное пространство	0xC1000000–E0FFFFFF, 0xA4000000–BFFFFFFF	64–960 Мб
x86, 1-гигабайтное системное пространство	0xC1000000–DBFFFFFF	64–432 Мб
x86, 1-гигабайтное системное пространство при наличии Terminal Services	0xC1000000–DCFFFFFF	64–448 Мб
x64	0xFFFFF98000000000–FFFFFA800000000064	64 Мб – 1024 Гб
IA64	0xE000000600000000–E00001060000000064	64 Мб – 1024 Гб

В таблице 11-2 перечислены системные переменные, которые содержат виртуальный размер и адрес системного кэша.

Таблица 11-2. *Системные переменные, хранящие виртуальный размер и адрес системного кэша*

Системная переменная	Описание
MmSystemCacheStart	Начальный виртуальный адрес кэша
MmSystemCacheEnd	Конечный виртуальный адрес кэша
MiSystemCacheStartExtra	Начальный виртуальный адрес дополнительной памяти кэша, превышающего 512 Мб (только на платформе x86)
MiSystemCacheEndExtra	Конечный виртуальный адрес дополнительной памяти кэша, превышающего 512 Мб (только на платформе x86)
MmSizeOfSystemCacheInPages	Максимальный размер системного кэша в страницах

ЭКСПЕРИМЕНТ: просмотр виртуального размера кэша

Виртуальный размер кэша не показывается каким-либо счетчиком производительности, так что единственный способ узнать его значение — получить содержимое переменной ядра *MmSizeOfSystemCacheInPages*:

```
kd> dd MmSizeOfSystemCacheInPages l 1
80539a8c  00020000
```

В этом примере использована x86-система под управлением Windows XP с параметром LargeSystemCache, равным 0; как видите, виртуальный размер кэша в такой системе составляет 0x20000 страниц. Поскольку на платформе x86 размер страниц равен 4 Кб, под виртуальный кэш выделено 512 Мб из 2-гигабайтного системного адресного пространства.

Размер рабочего набора кэша

Как уже упоминалось, одно из ключевых отличий архитектуры диспетчера кэша Windows от таковой в других операционных системах — делегирование управления физической памятью диспетчеру памяти. Ввиду этого размером кэша управляет уже имеющийся в операционной системе код, отвечающий за обработку расширения и усечения рабочего набора, а также за управление списками модифицированных и простаивающих страниц.

У системного кэша нет собственного рабочего набора — он использует единый системный набор, в который входят кэш данных, пул подкачиваемой памяти, а также подкачиваемый код Ntoskrnl и драйверов. Как упоминалось в главе 7, этот рабочий набор имеет внутреннее название *рабочий набор системного кэша*, но системный кэш является лишь одним из его элементов. Поэтому мы будем использовать термин «системный рабочий набор». Также в главе 7 мы обратили ваше внимание на то, что при присвоении параметру реестра LargeSystemCache значения 1 диспетчер памяти отдает пред-

почтение системному рабочему набору по сравнению с рабочими наборами процессов, выполняемых в системе.

Выяснить физический размер системного кэша, сравнить его с суммарным физическим размером системного рабочего набора, а также получить информацию об ошибках страниц для системного рабочего набора позволяют счетчики производительности или системные переменные, перечисленные в таблице 11-3.

Таблица 11-3. *Счетчики и системные переменные, хранящие информацию о физическом размере системного кэша и ошибках страниц*

Объект: счетчик	Системная переменная	Описание
Memory: System Cache Resident Bytes (Память: Резидентных байт системного кэша)	*MmSystemCachePage*	Физическая память, занятая под системный кэш
Memory: Cache Bytes (Память: Байт кэш-памяти)	*MmSystemCacheWs.WorkingSetSize*	Суммарный размер системного рабочего набора, включая кэш, пул подкачиваемой памяти, подкачиваемый код и проецируемые системой представления. Это не размер кэша, как можно подумать по названию счетчика!
Memory: Cache Bytes Peak [Память: Байт кэш-памяти (пик)]	*MmSystemCacheWs.Peak*	Пиковый размер системного рабочего набора
Memory: Cache Faults/Sec (Память: Ошибок кэш-памяти/сек)	*MmSystemCacheWs.PageFaultCount*	Число ошибок страниц, генерируемых системным рабочим набором (а не только кэшем)

ЭКСПЕРИМЕНТ: просмотр рабочего набора кэша

Как показано на листинге ниже, команда *!filecache* отладчика ядра выводит дамп информации о физической памяти, используемой кэшем, текущем и пиковом размерах рабочего набора, количестве действительных страниц, сопоставленных с представлениями, и, где это возможно, имена файлов, проецируемых на представления. (Драйверы файловых систем кэшируют метаданные с помощью безымянных файловых потоков.)

```
lkd> !filecache
***** Dump file cache******
  Reading and sorting VACBs ...
  Removed 657 nonactive VACBs, processing 1389 active VACBs ...
File Cache Information
  Current size 132308 kb
  Peak size    145764 kb
  1389 Control Areas
```

см. след. стр.

```
Skipping view @ c1040000 - no VACB, but PTE is valid!
  Loading file cache database (100% of 131072 PTEs)
  SkippedPageTableReads = 0
  File cache has 21510 valid pages

  Usage Summary (in Kb):
Control Valid Standby /Dirty Shared Locked Name
89be09c8   272       0     0      0  default_message_database.db
8a267ad8     4       0     0      0  $Directory
8a37d140  3504    1196     0      0  $Mft
8a35c928  4320    1676     0      0  outlook.pst
8a2acbb8     4       0     0      0  $Directory
8a42c908   564       0     0      0  $BitMap
8a42bd28    12       0     0      0  $Mft
8a29ed88     4       0     0      0  $Directory
8a467008    48       0     0      0  $Directory
8a42b828   124       0     0      0    No Name for File
8a467c58     8       0     0      0  $Directory
8a357f18   136      20     0      0  archive.pst
8a4a7248    52       0     0      0  $Directory
8a42a928   388       0     0      0  $Directory
8a457a10     4       0     0      0  $Directory
8a3f17a8     4       0     0      0  WIASERVC.LOG
89ede0e8     4       0     0      0  $Directory
8a491708 11900     136     0      0  $Mft
8a429fa0     4       0     0      0  $Directory
8a4318d8  2304   26624     0      0  $LogFile
8a438bc0     4       0     0      0  $Directory
8a4286a8     4       0     0      0  $Directory
8a491c50     8       0     0      0  $Directory
8a455de0     4       0     0      0  $Directory
8a3d0f00     4       0     0      0  $Directory
8a44d7a0  3420       0     0      0  $Mft
8a35ba10  1312       0    16      0  SHDOCVW.DLL
8a324130     4       0     0      0  $Directory
8a42c110     4       0     0      0  $Directory
8a41dd80     4       0     0      0  $Directory
88dfff38     4       0     0      0  $Directory
8a4a7640     4       0     0      0  $Directory
8a464d00     4       0     0      0  $Directory
```

Физический размер кэша

Хотя системный рабочий набор включает объем физической памяти, проецируемой на представления в виртуальном адресном пространстве кэша, он не обязательно отражает общий объем файловых данных, кэшируемых в физической памяти. Между этими двумя значениями нередко бывают рас-

хождения, потому что часть файловых данных может находиться в принадлежащем диспетчеру памяти списке простаивающих или модифицированных страниц.

Вспомните из главы 7, что при усечении рабочего набора или замене страниц диспетчер памяти может переместить измененные страницы из рабочего набора в список простаивающих или модифицированных страниц — в зависимости от того, куда должны быть записаны данные, содержащиеся на такой странице, перед ее повторным использованием — в страничный файл или в какой-то другой. Если бы у диспетчера памяти не было таких списков, то всякий раз, когда какой-нибудь процесс обращался бы к данным, ранее удаленным из его рабочего набора, диспетчеру памяти приходилось бы считывать их с диска. А так диспетчер памяти может просто вернуть нужную страницу в рабочий набор процесса (если она, конечно, присутствует в одном из этих списков). То есть списки служат кэшами данных из страничного файла, исполняемых образов или файлов данных. Значит, общий объем файловых данных, кэшируемых в системе, складывается не только из размера системного рабочего набора, но и из размеров списков простаивающих и модифицированных страниц.

Вот пример, иллюстрирующий, как диспетчер кэша способен привести к кэшированию в физической памяти гораздо большего объема файловых данных, чем может содержаться в системном рабочем наборе. Рассмотрим систему, выступающую в роли выделенного файл-сервера. В этой системе имеется 8 Гб физической памяти, и виртуальный размер кэша составляет 960 Мб (максимальный размер в x86-системах). Таким образом, предельный размер файловых данных, которые можно напрямую спроецировать в виртуальную память кэша, составляет 960 Мб. Клиентское приложение обращается к файловым данным на сервере через сеть. Драйвер файл-сервера (\Windows\System32\Drivers\Srv.sys) (см. главу 12) использует интерфейсы диспетчера кэша для чтения и записи файловых данных в интересах клиента. Если клиенты считывают несколько тысяч файлов, каждый размером по 1 Мб, диспетчеру кэша придется повторно использовать представления при проецировании 961-го файла. При последующих операциях чтения он будет отменять проецирование представлений для старых файлов и заново проецировать их для новых. Когда диспетчер кэша отменяет проецирование какого-либо представления, диспетчер памяти не отбрасывает файловые данные в рабочем наборе кэша, соответствующие этому представлению, а перемещает их в список простаивающих страниц. В отсутствие запросов на выделение физической памяти под любые другие задачи список простаивающих страниц может занимать почти всю физическую память за вычетом системного рабочего набора. Иначе говоря, практически все 8 Гб физической памяти сервера будут задействованы для кэширования файловых данных, как показано на рис. 11-6.

Рис. 11-6. *Пример использования почти всей физической памяти под файловый кэш*

Поскольку общий объем кэшируемых файловых данных складывается из размеров системного рабочего набора, списка модифицированных страниц и списка простаивающих страниц, а эти размеры контролируются диспетчером памяти, в каком-то смысле его можно назвать истинным диспетчером кэша. Подсистема диспетчера кэша просто предоставляет удобные интерфейсы для доступа к файловым данным через диспетчер памяти и определяет политики опережающего чтения (read-ahead) и отложенной записи (write-behind), которые влияют на то, какие данные диспетчер памяти будет удерживать в физической памяти.

Для более точного отображения полного объема файловых данных, кэшируемых в системе, диспетчер задач и Process Explorer предоставляют параметр System Cache (Системный кэш), отражающий суммарный размер системного рабочего набора и списков простаивающих и модифицированных страниц. Пример для Process Explorer представлен на рис. 11-7.

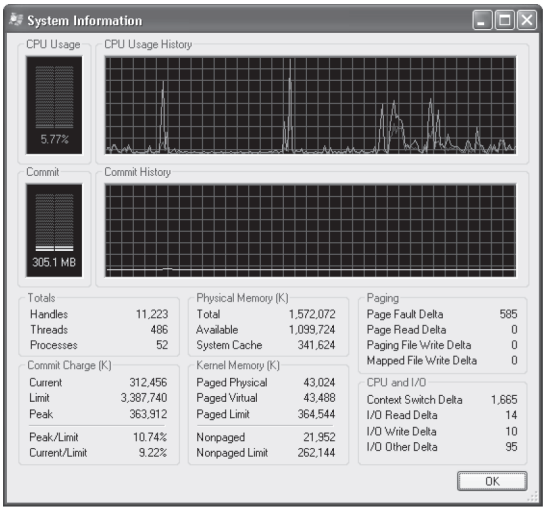


Рис. 11-7. *Диалоговое окно System Information в Process Explorer*

Структуры данных кэша

Для отслеживания кэшируемых файлов диспетчер кэша использует следующие структуры данных.

- Каждый 256-килобайтный слот системного кэша описывается VACB.
- У каждого отдельно открытого кэшируемого файла есть закрытая карта кэша с информацией, применяемой для контроля опережающего чтения.
- Каждый кэшируемый файл имеет общую структуру карты кэша, которая указывает на слоты системного кэша, содержащие проецируемые представления файла.

  Эти структуры и их взаимосвязи описываются в следующих разделах.

Общесистемные структуры данных кэша

Диспетчер кэша отслеживает состояние спроецированных на системный кэш представлений с помощью массива структур данных, называемых *блоками управления виртуальными адресами* (virtual address control blocks, VACB). При инициализации системы диспетчер кэша выделяет часть пула неподкачиваемой памяти под VACB, необходимые для описания системного кэша. Адрес массива VACB запоминается в переменной *CcVacbs*. Каждый VACB описывает одно 256-килобайтное представление в системном кэше (рис. 11-8). Структура VACB показана на рис. 11-9.

Рис. 11-8. *Системный массив VACB*

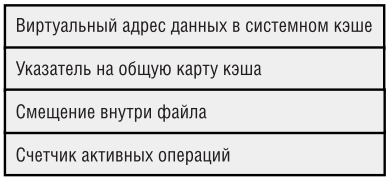


Рис. 11-9. *Структура VACB*

Как видно на рис. 11-9, в первом поле VACB содержится виртуальный адрес данных системного кэша. Второе поле является указателем на общую (совместно используемую) структуру карты кэша, которая идентифицирует кэшируемый файл. Третье поле определяет смещение начала представления (внутри файла). Наконец, VACB содержит счетчик ссылок на представление, т. е. число активных операций чтения или записи над данным представлением. При выполнении операции ввода-вывода над файлом счетчик ссылок VACB увеличивается на 1, а по окончании такой операции уменьшается на 1. Когда счетчик ссылок не равен 0, VACB считается активным. В случае обращения к метаданным файловой системы счетчик активных операций отражает число драйверов файловых систем, которые владеют заблокированными в памяти страницами данного представления.

Структуры данных кэша, индивидуальные для каждого файла

Каждому открытому описателю файла соответствует объект «файл» (см. главу 9). Если файл кэшируется, его объект «файл» указывает на структуру *закрытой карты кэша* (private cache map), которая содержит два адреса, по которым в последний раз происходило чтение данных. Кроме того, все закрытые карты кэша для открытых экземпляров файла связаны друг с другом.

У каждого кэшируемого файла (в противоположность объекту «файл») есть структура *общей карты кэша* (shared cache map), которая описывает состояние кэшируемого файла, в том числе его размер и (из соображений безопасности) длину его действительных данных. (О назначении поля длины действительных данных файла см. раздел «Кэширование с обратной записью и отложенная запись» далее в этой главе). Общая карта кэша также указывает на объект-раздел (поддерживаемый диспетчером памяти и описывающий проекцию файла на виртуальную память), список закрытых карт памяти, сопоставленных с этим файлом, и все VACB, описывающие представления файлов, проецируемые в данный момент на системный кэш. Взаимосвязи между этими структурами данных показаны на рис. 11-10.

При запросе на чтение данных из какого-либо файла диспетчер кэша должен ответить на два вопроса.

1. Находится ли файл в кэше?
2. Если да, то какие VACB (если таковые есть) ссылаются на запрошенный адрес?

Иначе говоря, диспетчер кэша должен выяснить, проецируется ли представление файла (с нужным смещением) на системный кэш. Если ни один VACB не содержит нужное смещение в файле, запрошенные данные в настоящий момент не проецируются на системный кэш.

Для учета представлений данного файла, проецируемых на системный кэш, диспетчер кэша поддерживает массив указателей на VACB — *массив индексов VACB* (VACB index array). Первый элемент массива индексов VACB ссылается на первые 256 Кб файла, второй — на следующие 256 Кб и т. д.

Схема на рис. 11-11 иллюстрирует четыре раздела из трех файлов, проецируемых в данный момент на системный кэш.

Когда процесс обращается к файлу по заданному адресу, диспетчер кэша ищет подходящий элемент в массиве индексов VACB для этого файла, чтобы определить, проецируются ли на кэш запрошенные данные. Если элемент массива отличен от 0 (и, следовательно, содержит указатель на VACB), нужная область файла находится в кэше. VACB в свою очередь указывает на адрес, по которому на системный кэш проецируется представление файла. А если элемент массива равен 0, диспетчер кэша должен найти в системном кэше свободный слот (а значит, свободный VACB) для проецирования необходимого представления.

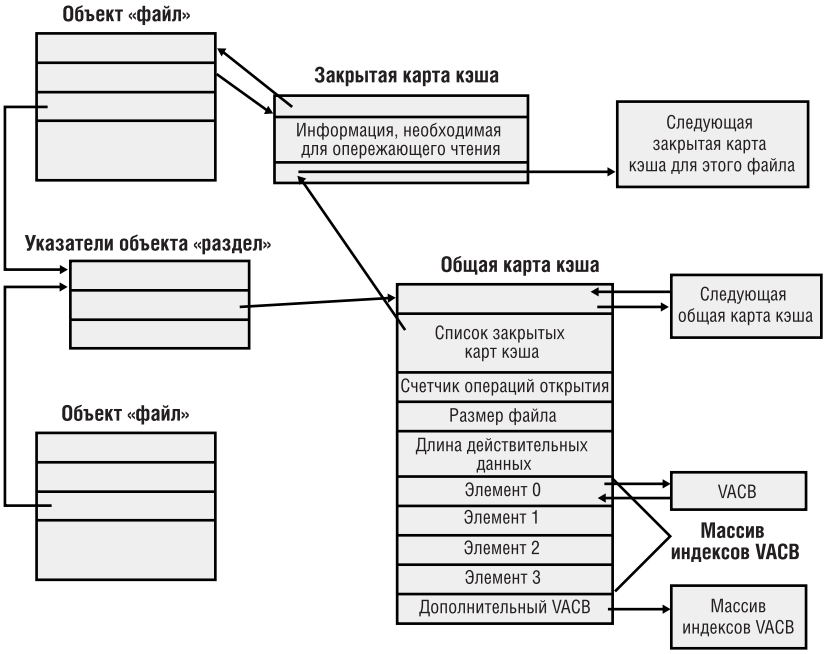


Рис. 11-10. *Структуры данных кэша, индивидуальные для файлов*

Для оптимизации своего размера общая карта кэша содержит массив индексов VACB из 4 элементов. Поскольку каждый VACB описывает 256 Кб, элементы этого компактного массива индексов фиксированного размера могут указывать на элементы массива VACB, которые в совокупности способны описывать файл размером до 1 Мб. Если размер файла превышает 1 Мб, из неподкачиваемого пула выделяется память под отдельный массив индексов VACB; его размер определяется делением размера файла на 256 Кб с последующим округлением результата до ближайшего большего целого значения. После этого общая карта кэша указывает на данную структуру.

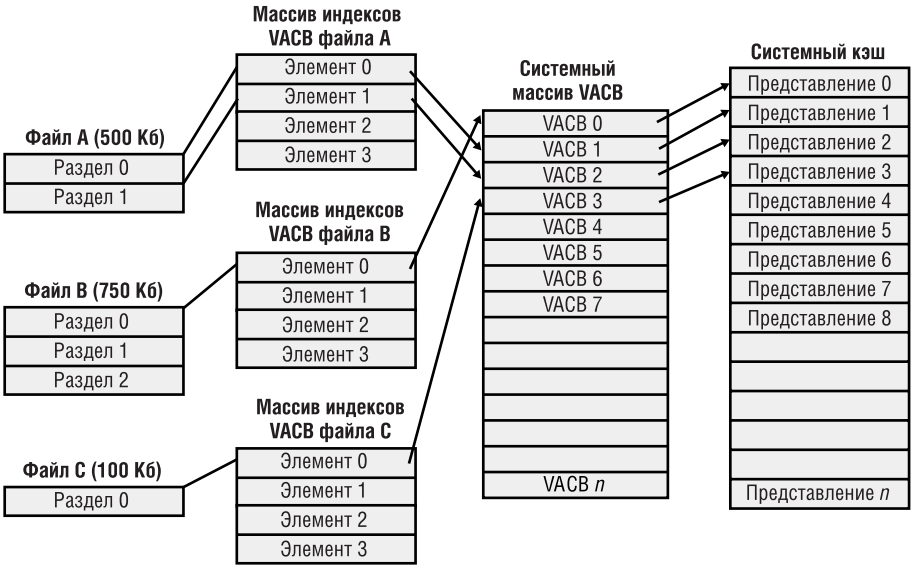


Рис. 11-11. *Массивы индексов VACB*

Если длина файла превышает 32 Мб, то для еще большей оптимизации массив индексов VACB, созданный в пуле неподкачиваемой памяти, становится разреженным многоуровневым массивом индексов (sparse multilevel index array), в котором каждый массив индексов состоит из 128 элементов. Число уровней, необходимых для файла, вычисляется по формуле:

```
(Разрядность значения, отражающего длину файла - 18) / 7
```

Полученное значение надо округлить до ближайшего большего целого. Число 18 в уравнении обусловлено тем, что VACB представляет 256 Кб, а 256 Кб — это 2^{18}. Наконец, число 7 присутствует в уравнении потому, что каждый уровень массива состоит из 128 элементов, а 128 — это 2^7. Следовательно, файл максимальной длины, которая может быть описана как 2^{63} (максимальный размер, поддерживаемый диспетчером кэша), потребует всего 7 уровней. Массив является разреженным, так как диспетчер кэша создает ветви лишь для активных представлений на самом низком уровне массива индексов. На рис. 11-12 показан пример многоуровневого массива VACB для разреженного файла, размер которого требует для описания 3 уровня.

Такая схема нужна для эффективной обработки разреженных файлов, которые могут достигать очень больших размеров и в которых лишь малая часть может быть занята действительными данными; поэтому в массиве выделяется ровно столько места, сколько нужно для проецируемых в данный момент представлений файла. Например, разреженный файл размером 32 Гб, у которого только 256 Кб проецируются на виртуальное адресное пространство кэша, потребует массив VACB с тремя массивами индексов, поскольку лишь одна ветвь массива имеет проекцию, а для файла длиной 32 Гб (2^{35} байтов) нужен трехуровневый массив. Если бы диспетчер кэша не оптимизировал

многоуровневые массивы VACB, для этого файла пришлось бы создать массив VACB со 128 000 элементов, что эквивалентно 1000 массивам индексов.

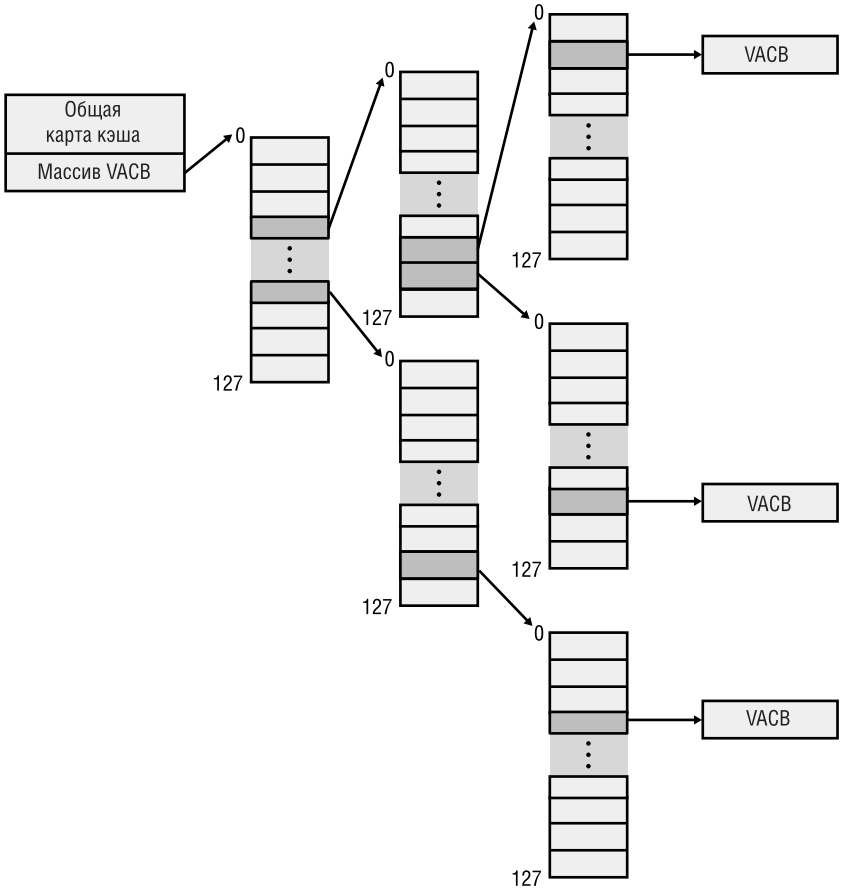


Рис. 11-12. *Многоуровневые массивы VACB*

ЭКСПЕРИМЕНТ: просмотр общей и закрытых карт кэша

Команда *dt* отладчика ядра позволяет увидеть определения структур данных общей и закрытой карт кэша в работающей системе. Во-первых, выполните команду *!filecache* и найдите запись в выводе VACB с именем известного вам файла. В нашем примере таковым будет справочный файл из Debugging Tools for Windows:

```
8653c828  120  160   0   0 debugger.chm
```

Первый адрес указывает местонахождение структуры данных области управления (control area), с помощью которой диспетчер памяти отслеживает диапазон адресов. (Более подробные сведения см. в главе 7.) В области управления хранится указатель на объект «файл», со-

см. след. стр.

ответствующий представлению в кэше. Объект «файл» идентифицирует экземпляр открытого файла — в данном случае справочного файла из Debugging Tools for Windows. Теперь, чтобы увидеть структуру области управления, введите следующую команду с адресом идентифицированного вами элемента в этой области:

```
lkd> !ca 8653c828

ControlArea @8653c828
  Segment:    e1bce428    Flink              0   Blink          0
  Section Ref        1    Pfn Ref           46   Mapped Views   4
  User Ref           0    WaitForDel         0   Flush Count    0
  File Object 85d927a8    ModWriteCount      0   System Views   0

  Flags (8008080) File WasPurged Accessed

  File: \Program Files\Debugging Tools for Windows\debugger.chm
```

Потом изучите объект «файл», на который ссылается область управления:

```
lkd> dt _File_object 85d927a8
nt!_FILE_OBJECT
   +0x000 Type             : 5
   +0x002 Size             : 112
   +0x004 DeviceObject     : 0x86b78e30
   +0x008 Vpb              : 0x86b90d80
   +0x00c FsContext        : 0xe3b629e0
   +0x010 FsContext2       : 0xe3b62b38
   +0x014 SectionObjectPointer : 0x85e7a334
   +0x018 PrivateCacheMap  : 0x862f5a50
   +0x01c FinalStatus      : 0
   +0x020 RelatedFileObject : (null)
   +0x024 LockOperation    : 0 ''
...
```

Закрытая карта кэша находится по смещению 0x18:

```
lkd> dt _private_cache_map 0x862f5a50
nt!_PRIVATE_CACHE_MAP
   +0x000 NodeTypeCode     : 766
   +0x000 Flags            : _PRIVATE_CACHE_MAP_FLAGS
   +0x000 UlongFlags       : 0x2fe
   +0x004 ReadAheadMask    : 0xffff
   +0x008 FileObject       : 0x85d927a8
   +0x010 FileOffset1      : _LARGE_INTEGER 0x10b28
   +0x018 BeyondLastByte1  : _LARGE_INTEGER 0x12206
   +0x020 FileOffset2      : _LARGE_INTEGER 0x10b28
   +0x028 BeyondLastByte2  : _LARGE_INTEGER 0x12206
   +0x030 ReadAheadOffset  : [2] _LARGE_INTEGER 0x0
```

```
   +0x040 ReadAheadLength  : [2] 0
   +0x048 ReadAheadSpinLock : 0
   +0x04c PrivateLinks     : _LIST_ENTRY [ 0x862f5a10 - 0x862f5a10 ]
```

Наконец, вы можете найти общую карту кэша в структуре SectionObjectPointers объекта «файл», а затем просмотреть ее содержимое:

```
lkd> dt _Section_object_pointers 0x85e7a334
nt!_SECTION_OBJECT_POINTERS
   +0x000 DataSectionObject : 0x8653c828
   +0x004 SharedCacheMap   : 0x862f5978
   +0x008 ImageSectionObject : (null)

lkd> dt _shared_cache_map 0x862f5978  nt!_SHARED_CACHE_MAP
   +0x000 NodeTypeCode     : 767
   +0x002 NodeByteSize     : 304
   +0x004 OpenCount        : 1
   +0x008 FileSize         : _LARGE_INTEGER 0x19970a
   +0x010 BcbList          : _LIST_ENTRY [ 0x862f5988 - 0x862f5988 ]
   +0x018 SectionSize      : _LARGE_INTEGER 0x1c0000
   +0x020 ValidDataLength  : _LARGE_INTEGER 0x19970a
   +0x028 ValidDataGoal    : _LARGE_INTEGER 0x19970a
   +0x030 InitialVacbs     : [4] (null)
   +0x040 Vacbs            : 0x85d42c00  -> 0x86bba610
   +0x044 FileObject       : 0x85d927a8
   +0x048 ActiveVacb       : 0x86bba610
...
```

Интерфейсы файловых систем

При первом обращении к файловым данным для чтения или записи драйвер файловой системы должен определить, проецируются ли нужные части файла на системный кэш. Если нет, драйвер файловой системы должен вызвать функцию *CcInitializeCacheMap* для подготовки индивидуальных для каждого файла структур данных кэша.

Далее драйвер файловой системы вызывает одну из нескольких функций для доступа к данным файла. Существует три основных метода доступа к кэшируемым данным, каждый из которых рассчитан на применение в определенной ситуации:

- копирование (copy method) — пользовательские данные копируются между буферами кэша в системном пространстве и буфером процесса в пользовательском пространстве;
- проецирование и фиксация (mapping and pinning method) — данные считываются и записываются прямо в буферы кэша по виртуальным адресам;
- обращение к физической памяти (physical memory access method) — данные считываются и записываются прямо в буферы кэша по физическим адресам.

Чтобы избежать бесконечного цикла при обработке диспетчером памяти ошибки страницы, драйверы файловых систем должны поддерживать два варианта чтения файлов — с кэшированием и без. В таких случаях диспетчер памяти вызывает файловую систему для получения данных из файла (через драйвер устройства) и запрашивает операцию чтения без кэширования, устанавливая в IRP флаг «no cache».

Рис. 11-13 иллюстрирует типичное взаимодействие между диспетчером кэша, диспетчером памяти и драйверами файловой системы в ответ на пользовательские операции файлового ввода-вывода (чтения или записи). Диспетчер кэша вызывается файловой системой через интерфейсы копирования (функции *CcCopyRead* и *CcCopyWrite*). Чтобы обработать, например, операцию чтения, инициированную через *CcFastCopyRead* или *CcCopyRead*, диспетчер кэша создает представление в кэше для проецирования части запрошенного файла и считывает файловые данные в пользовательский буфер, копируя их из представления. Операция копирования генерирует ошибки страниц по мере обращения к каждой ранее недействительной странице в представлении, и в ответ диспетчер памяти инициирует ввод-вывод без кэширования, используя драйвер файловой системы для выборки данных, соответствующих части файла, спроецированной на ту страницу, которая оказалась недействительной.

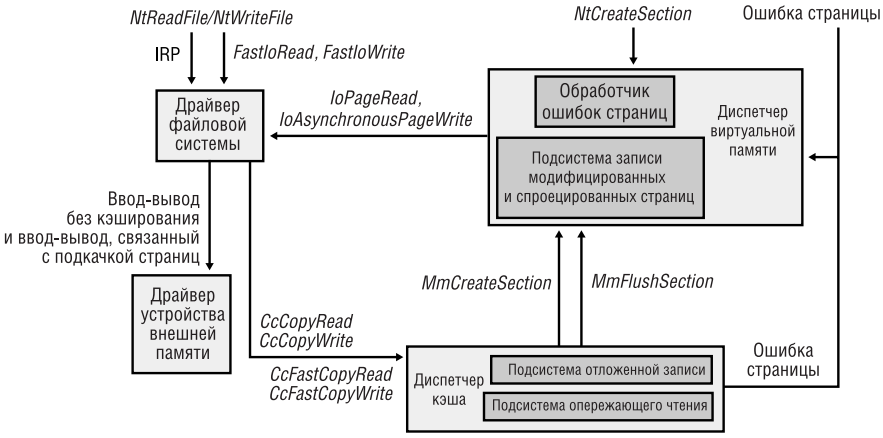


Рис. 11-13. *Взаимодействие файловой системы с диспетчерами кэша и памяти*

В следующих трех разделах мы рассмотрим все три ранее упомянутых механизма доступа к кэшу, их предназначение и принципы использования.

Копирование данных в кэш и из него

Поскольку системный кэш находится в системном пространстве, он проецируется на адресное пространство каждого процесса. Однако, как и любые другие страницы системного пространства, страницы кэша недоступны в пользовательском режиме, поскольку иначе в защите появилась бы потенциальная дыра. (Например, процесс, не имеющий соответствующих прав,

мог бы считать данные из файла, который находится в какой-либо части системного кэша.) Таким образом, операции чтения и записи пользовательских приложений в файлы должны обслуживаться процедурами режима ядра, которые копируют данные между буферами кэша в системном пространстве и буферами приложения, расположенными в адресном пространстве процесса. Функции, которые драйверы файловой системы могут использовать для выполнения этих операций, перечислены в таблице 11-4.

Таблица 11-4. *Функции режима ядра для копирования данных в кэш и из него*

Функция	Описание
CcCopyRead	Копирует заданный диапазон байтов из системного кэша в пользовательский буфер
CcFastCopyRead	Более быстрый вариант *CcCopyRead*, но ограничен использованием 32-разрядных смещений внутри файла и синхронного чтения
CcCopyWrite	Копирует заданный диапазон байтов из пользовательского буфера в системный кэш
CcFastCopyWrite	Более быстрый вариант *CcCopyWrite*, но ограничен использованием 32-разрядных смещений внутри файла и синхронной (не сквозной) записи (применим только в NTFS)

Активность операций чтения из кэша можно увидеть через счетчики производительности и системные переменные, представленные в таблице 11-5.

Таблица 11-5. *Счетчики и системные переменные, отражающие активность операций чтения из кэша*

Объект: счетчик	Системная переменная	Описание
Cache: Copy Read Hits % (Кэш: % попаданий при чтении с копированием)	(*CcCopyReadWait* + *CcCopyReadNoWait*) / (*CcCopyReadWait* +(*CcCopyReadWaitMiss* + *CcCopyReadNoWait*) + *CcCopyReadNoWaitMiss*)	Процентная доля операций чтения с копированием из кэшированных частей файлов (чтение с копированием все равно может вызывать подкачку — счетчик Memory: Cache Faults/Sec сообщает об интенсивности генерации ошибок страниц для системного рабочего набора, но включает как аппаратные, так и программные ошибки страниц и поэтому не отражает реальную интенсивность ввода-вывода из-за подкачки, вызванной ошибками страниц кэша)
Cache: Copy Reads/Sec (Кэш: Чтений с копированием/сек)	*CcCopyReadWait* + *CcCopyReadNoWait*	Суммарное число операций чтения с копированием из кэша

см. след. стр.

Таблица 11-5. (*окончание*)

Объект: счетчик	Системная переменная	Описание
Cache: Sync Copy Reads/Sec (Кэш: Синхронных чтений с копированием/сек)	*CcCopyReadWait*	Число синхронных операций чтения с копированием из кэша
Cache: Async Copy Reads/Sec (Кэш: Асинхронных чтений с копированием/сек)	*CcCopyReadNoWait*	Число асинхронных операций чтения с копированием из кэша

Кэширование с применением интерфейсов проецирования и фиксации

По мере чтения и записи данных в дисковые файлы пользовательскими приложениями драйверы файловых систем должны считывать и записывать данные, описывающие сами файлы (метаданные, или данные о структуре тома). Так как драйверы файловых систем выполняются в режиме ядра, они могут модифицировать данные непосредственно в системном кэше при условии уведомления об этом диспетчера кэша. Для поддержки такой оптимизации диспетчер кэша предоставляет функции, перечисленные в таблице 11-6. Эти функции позволяют драйверам файловых систем находить в виртуальной памяти нужные метаданные и напрямую модифицировать их без использования промежуточных буферов.

Таблица 11-6. *Функции поиска адресов метаданных*

Функция	Описание
CcMapData	Проецирует диапазон байтов для чтения
CcPinRead	Проецирует диапазон байтов для чтения/записи и фиксирует его
CcPreparePinWrite	Проецирует и фиксирует диапазон байтов для записи (чтение недопустимо)
CcPinMappedData	Фиксирует ранее спроецированный буфер
CcSetDirtyPinnedData	Уведомляет диспетчер кэша о модификации данных
CcUnpinData	Отменяет фиксацию страниц, после чего они могут быть удалены из памяти

Если драйверу файловой системы нужно считать метаданные из кэша, он вызывает интерфейс диспетчера кэша, отвечающий за проецирование, чтобы получить виртуальный адрес требуемых данных. Диспетчер кэша подгружает в память все запрошенные страницы и возвращает управление драйверу файловой системы. После этого драйвер может напрямую обращаться к данным.

Если драйверу файловой системы необходимо модифицировать страницы кэша, он вызывает сервисы диспетчера кэша, отвечающие за фиксацию модифицируемых страниц в памяти. На самом деле эти страницы не блокируются в памяти (как это происходит в тех случаях, когда драйвер устрой-

ства блокирует страницы для передачи данных с использованием прямого доступа к памяти). По большей части драйвер файловой системы помечает их поток метаданных как «no write», сообщая подсистеме записи модифицированных страниц диспетчера памяти (см. главу 7) не сбрасывать страницы на диск до тех пор, пока не будет явно указано иное. После отмены фиксации страниц диспетчер кэша сбрасывает на диск все измененные страницы и освобождает представление кэша, которое было занято метаданными.

Интерфейсы проецирования и фиксации решают одну сложную проблему реализации файловых систем — управление буферами. В отсутствие возможности прямых операций над кэшированными метаданными файловая система была бы вынуждена предугадывать максимальное число буферов, которое понадобится ей для обновления структуры тома. Обеспечивая файловой системе прямой доступ к ее метаданным и их изменение непосредственно в кэше, диспетчер кэша устраняет потребность в буферах и просто обновляет структуру тома в виртуальной памяти, предоставленной диспетчером памяти. Единственным ограничением файловой системы в этом случае является объем доступной памяти.

Вы можете наблюдать за интенсивностью операций, связанных с фиксацией и проецированием в кэше, с помощью счетчиков производительности и системных переменных, перечисленных в таблице 11-7.

Таблица 11-7. *Счетчики и системные переменные для наблюдения за интенсивностью операций фиксации и проецирования*

Объект: счетчик	Системная переменная	Описание
Cache: Data Map Hits % (Кэш: % попаданий при отображении данных)	(*CcMapDataWait* + *CcMapDataNoWait*) / (*CcMapDataWait* + *CcMapDataNoWait*) + (*CcMapDataWaitMiss* + *CcMapDataNoWaitMiss*)	Процентная доля операций проецирования данных применительно к кэшированным частям файлов (чтение с копированием все равно может вызывать подкачку)
Cache: Data Maps/Sec (Кэш: Отображений данных/сек)	*CcMapDataWait* + *CcMapDataNoWait*	Суммарное число операций проецирования данных из кэша
Cache: Sync Data Maps/Sec (Кэш: Синхронных отображений данных/сек)	*CcMapDataWait*	Число операций синхронного проецирования данных из кэша
Cache: Async Data Maps/Sec (Кэш: Асинхронных отображений данных/сек)	*CcMapDataNoWait*	Число операций асинхронного проецирования данных из кэша
Cache: Data Map Pins/Sec (Кэш: Фиксаций при отображении данных/сек)	*CcPinMappedDataCount*	Число запросов на фиксацию спроецированных данных

см. след. стр.

Таблица 11-7. (*окончание*)

Объект: счетчик	Системная переменная	Описание
Cache: Pin Read Hits % (Кэш: % попаданий фиксации при чтении)	(*CcPinReadWait* + *CcPinReadNoWait*) / (*CcPinReadWait* + *CcPinReadNoWait*) + (*CcPinReadWaitMiss* + *CcPinReadNoWaitMiss*)	Процентная доля операций чтения применительно к фиксированным в кэше частям файлов (чтение с копированием все равно может вызывать подкачку)
Cache: Pin Reads/Sec (Кэш: Фиксаций при чтении/сек)	*CcPinReadWait* + *CcPinReadNoWait*	Суммарное число операций чтения из фиксированных страниц кэша
Cache: Sync Pin Reads/Sec (Кэш: Синхронных фиксаций при чтении/сек)	*CcPinReadWait*	Число синхронных операций чтения из фиксированных страниц кэша
Cache: Async Pin Reads/Sec (Кэш: Асинхронных фиксаций при чтении/сек)	*CcPinReadNoWait*	Число асинхронных операций чтения из фиксированных страниц кэша

Кэширование с применением прямого доступа к памяти

В дополнение к интерфейсам проецирования и фиксации, используемым при прямом обращении к кэшированным метаданным, диспетчер кэша предоставляет третий интерфейс — *прямой доступ к памяти* (direct memory access, DMA). Функции DMA применяются для чтения или записи страниц кэша без промежуточных буферов, например сетевой файловой системой при передаче данных по сети.

Интерфейс DMA возвращает файловой системе физические адреса кэшируемых пользовательских данных (а не виртуальные, которые возвращаются интерфейсами проецирования и фиксации), и эти адреса могут быть использованы для прямой передачи данных из физической памяти на сетевое устройство. Хотя при передаче небольших порций данных (1–2 Кб) можно пользоваться обычными интерфейсами копирования на основе буферов, при передаче больших объемов данных интерфейс DMA значительно повышает быстродействие сетевого сервера, обрабатывающего файловые запросы от удаленных систем.

Для описания ссылок на физическую память служит список дескрипторов памяти (memory descriptor list, MDL) (см. главу 7). DMA-интерфейс диспетчера кэша состоит их четырех функций (таблица 11-8).

Таблица 11-8. *Функции DMA-интерфейса*

Функция	Описание
CcMdlRead	Возвращает MDL, описывающий заданный диапазон байтов
CcMdlReadComplete	Освобождает MDL
CcMdlWrite	Возвращает MDL, описывающий заданный диапазон байтов (возможно, содержащий нулевые значения)
CcMdlWriteComplete	Освобождает MDL и отмечает диапазон для записи

Вы можете исследовать активность, связанную с MDL-чтением из кэша, через счетчики производительности или системные переменные, перечисленные в таблице 11-9.

Таблица 11-9. *Счетчики и системные переменные, позволяющие наблюдать за интенсивностью операций MDL-чтения из кэша*

Объект: счетчик	Системная переменная	Описание
Cache: MDL Read Hits % (Кэш: % попаданий чтений MDL)	(*CcMdlReadWait* + *CcMdlReadNoWait*) / (*CcMdlReadWait* + *CcMdlReadNoWait*) + (*CcMdlReadWaitMiss* + (*CcMdlReadWaitMiss* + *CcMdlReadNoWaitMiss*)	Процентная доля операций MDL-чтения применительно к кэшированным частям файлов (ссылки на страницы, разрешаемые MDL-чтением, все равно могут вызывать подкачку)
Cache: MDL Reads/Sec (Кэш: Чтений MDL/сек)	*CcMdlReadWait* + *CcMdlReadNoWait*	Суммарное число операций MDL-чтения из кэша
Cache: Sync MDL Reads/Sec (Кэш: Синхронных чтений MDL/сек)	*CcMdlReadWait*	Число операций синхронного MDL-чтения из кэша
Cache: Async MDL Reads/Sec (Кэш: Асинхронных чтений MDL/сек)	*CcMdlReadNoWait*	Число операций асинхронного MDL-чтения из кэша

Быстрый ввод-вывод

Операции чтения и записи, выполняемые над кэшируемыми файлами, по возможности обрабатываются с применением высокоскоростного механизма — *быстрого ввода-вывода* (fast I/O). Как уже говорилось в главе 9, быстрый ввод-вывод обеспечивает чтение и запись кэшируемых файлов без генерации IRP. При использовании этого механизма диспетчер ввода-вывода вызывает процедуру быстрого ввода-вывода, принадлежащую драйверу файловой системы, и определяет, можно ли удовлетворить ввод-вывод непосредственно из кэша без генерации IRP.

Поскольку диспетчер кэша в архитектуре системы размещается поверх подсистемы виртуальной памяти, драйверы файловых систем могут использовать этот диспетчер для доступа к данным путем простого копирования их

в страницы (или из страниц), проецируемые на тот файл, на который ссылается пользовательская программа, без генерации IRP.

Быстрый ввод-вывод возможен не всегда. Например, первая операция чтения или записи требует подготовки файла к кэшированию (его проецирования на кэш и создания структур данных кэша, описанных в разделе «Структуры данных кэша» ранее в этой главе). Быстрый ввод-вывод не применяется и в том случае, если вызывающий поток указывает асинхронное чтение или запись, поскольку этот поток может быть приостановлен в ходе операций ввода-вывода, связанных с подкачкой и необходимых для копирования буферов в системный кэш (и из него), и фактически синхронного выполнения запрошенной операции асинхронного ввода-вывода. Однако даже при синхронном вводе-выводе драйвер файловой системы может решить, что обработка запрошенной операции по механизму быстрого ввода-вывода недопустима, если, например, в нужном файле заблокирован какой-то диапазон байтов (в результате вызова Windows-функции *LockFile*). Поскольку диспетчер кэша не знает, какие части и каких файлов блокированы, драйвер файловой системы должен проверить возможность чтения или записи запрошенных данных, а это требует генерации IRP. Алгоритм принятия решений показан на рис. 11-14.

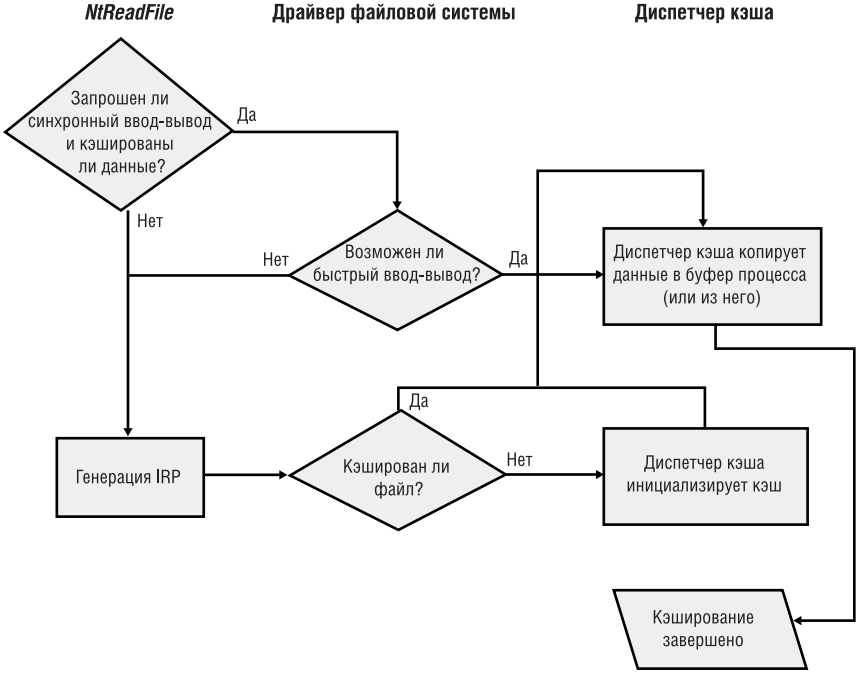


Рис. 11-14. *Алгоритм принятия решений по быстрому вводу-выводу*

Обслуживание чтения или записи с использованием быстрого ввода-вывода включает следующие операции.

1. Поток выполняет операцию чтения или записи.
2. Если файл кэшируется и указан синхронный ввод-вывод, запрос передается входной точке быстрого ввода-вывода драйвера файловой системы. Если файл не кэшируется, драйвер файловой системы готовит файл к кэшированию, чтобы выполнить следующий запрос на чтение или запись за счет быстрого ввода-вывода.
3. Если процедура драйвера файловой системы, отвечающая за быстрый ввод-вывод, определяет, что быстрый ввод-вывод возможен, она вызывает процедуру чтения или записи диспетчера кэша для прямого доступа к данным кэша. (Если быстрый ввод-вывод невозможен, драйвер файловой системы возвращает управление подсистеме ввода-вывода, которая затем генерирует IRP и в конечном счете вызывает в файловой системе обычную процедуру чтения.)
4. Диспетчер кэша транслирует переданное смещение в файле в виртуальный адрес данных в кэше.
5. При операциях чтения диспетчер кэша копирует данные из кэша в буфер процесса, а при операциях записи — из буфера процесса в кэш.
6. Выполняется одна из следующих операций:
   - при операциях чтения из файла, при открытии которого не был установлен флаг FILE_FLAG_RANDOM_ACCESS, в закрытой карте кэша вызывающего потока обновляется информация, необходимая для опережающего чтения;
   - при операциях записи устанавливается бит изменения у всех модифицированных страниц кэша, чтобы подсистема отложенной записи сбросила эти страницы на диск;
   - для файлов, требующих сквозной записи, все измененные данные немедленно сбрасываются на диск.

ПРИМЕЧАНИЕ Быстрый ввод-вывод возможен не только в тех случаях, когда запрошенные данные уже находятся в физической памяти. Как видно из пп. 5 и 6 предыдущего списка, диспетчер кэша просто обращается по виртуальным адресам уже открытого файла, где он предполагает найти нужные данные. Если происходит промах кэша, диспетчер памяти динамически подгружает эти данные в физическую память.

Счетчики производительности и системные переменные, перечисленные в таблице 11-10, позволяют наблюдать за операциями быстрого ввода-вывода в системе.

Таблица 11-10. *Счетчики и системные переменные, отражающие активность быстрого ввода-вывода*

Объект: счетчик	Системная переменная	Описание
Cache: Sync Fast Reads/Sec (Кэш: Синхронных быстрых чтений/сек)	*CcFastReadWait*	Число операций синхронного чтения, обработанных как запросы быстрого ввода-вывода
Cache: Async Fast Reads/Sec (Кэш: Асинхронных быстрых чтений/сек)	*CcFastReadNoWait*	Число операций асинхронного чтения, обработанных как запросы быстрого ввода-вывода (это значение всегда равно 0, так как асинхронное быстрое чтение в Windows не выполняется)
Cache: Fast Read Resource Misses/Sec (Кэш: Промахов ресурса быстрого чтения/сек)	*CcFastReadResourceMiss*	Число операций быстрого ввода-вывода, не выполненных из-за конфликтов ресурсов (это возможно только при использовании FAT — в NTFS этого не бывает)
Cache: Fast Read Not Possibles/Sec (Кэш: Не осуществленных быстрых чтений/сек)	*CcFastReadNotPossible*	Число операций быстрого ввода-вывода, которые не удалось выполнить (по решению драйвера файловой системы, например, из-за того, что какой-то диапазон байтов в файле блокирован)

Опережающее чтение и отложенная запись

Здесь вы увидите, как диспетчер кэша реализует чтение и запись файловых данных в интересах драйверов файловых систем. Учтите, что диспетчер кэша участвует в файловом вводе-выводе только при открытии файла без флага FILE_FLAG_NO_BUFFERING и последующем чтении или записи через Windows-функции ввода-вывода (например, функции *ReadFile* и *WriteFile*). Кроме того, диспетчер кэша не имеет дела с проецируемыми файлами, а также с файлами, открытыми с флагом FILE_FLAG_NO_BUFFERING.

Интеллектуальное опережающее чтение

Для реализации *интеллектуального опережающего чтения* (intelligent read-ahead) диспетчер кэша использует принцип пространственной локальности (spatial locality); исходя из данных, которые вызывающий процесс считывает в данный момент, диспетчер кэша пытается предсказать, какие данные тот будет считывать в следующий раз. Поскольку системный кэш опирается на использование виртуальных адресов, непрерывных для конкретного файла, их непрерывность в физической памяти не имеет значения. Реализация опережающего чтения файлов при кэшировании на основе логических блоков была бы гораздо сложнее и потребовала бы тесной координации между драйверами файловых систем и кэшем, поскольку такая система кэширования опирается на относительные позиции затребованных данных на диске,

а файлы вовсе не обязательно хранятся в непрерывных областях диска. Активность, связанную с опережающим чтением, можно исследовать с помощью счетчика производительности Cache: Read Aheads/Sec (Кэш: Упреждающих чтений/сек) или системной переменной *CcReadAheadIos*.

Считывание следующего блока файла, к которому происходит последовательное обращение, дает очевидные преимущества. Чтобы распространить эти преимущества и на случаи произвольного (прямого) доступа к данным (в направлении вперед или назад), диспетчер кэша запоминает последние два запроса на чтение в закрытой карте кэша, сопоставленной с описателем файла, к которому обращается программа. Этот метод называется *асинхронным опережающим чтением с хронологией*. Диспетчер кэша пытается выявить какую-то закономерность в операциях прямого чтения вызывающей программы. Например, если вызывающая программа считывает сначала страницу 4000, затем 3000, диспетчер кэша предполагает, что в следующий раз будет затребована страница 2000, и заблаговременно считывает ее в кэш.

ПРИМЕЧАНИЕ Хотя предсказание возможно лишь на основе последовательности из трех операций чтения минимум, в закрытой карте кэша запоминаются только две из них.

Чтобы еще больше повысить эффективность опережающего чтения, Windows-функция *CreateFile* поддерживает флаг последовательного доступа к файлу, FILE_FLAG_SEQUENTIAL_SCAN. Если этот флаг задан, диспетчер кэша не ведет хронологию чтения для предсказаний, выполняя вместо этого последовательное опережающее чтение. Но по мере считывания файла в рабочий набор кэша диспетчер кэша удаляет проекции неактивных представлений файла и командует диспетчеру памяти переместить страницы, принадлежавшие удаленным проекциям, в начало списка простаивающих или модифицированных страниц (если страницы изменены), чтобы впоследствии их можно было быстро использовать повторно. Он также заранее считывает двукратный объем данных (например, 128 Кб вместо 64 Кб). По мере того как вызывающий поток продолжает считывать данные, диспетчер кэша считывает дополнительные блоки данных, всегда опережая вызывающий поток на один блок, равный текущему запрошенному.

В этом случае опережающее чтение выполняется диспетчером кэша асинхронно, так как это делается в контексте отдельного потока, выполняемого параллельно с вызывающим потоком. Когда диспетчер кэша вызывается для выдачи кэшированных данных, он сначала обращается к запрошенной виртуальной странице, чтобы удовлетворить запрос, а затем ставит в очередь системного рабочего потока еще один запрос на ввод-вывод для выборки дополнительной порции данных. Далее рабочий поток выполняется в фоновом режиме и считывает дополнительные данные, упреждая следующий запрос вызывающего потока. Заранее считанные страницы загружаются в память параллельно выполнению пользовательской программы, так что на момент выдачи ее потоком очередного запроса эти данные уже находятся в памяти.

В случае приложений, для которых невозможно предсказать схему чтения данных, функция *CreateFile* предусматривает флаг FILE_FLAG_RANDOM_ACCESS. Этот флаг запрещает диспетчеру кэша предсказание адресов следующих операций чтения и тем самым отключает опережающее чтение. Этот флаг также предотвращает агрессивное удаление диспетчером кэша проекций представлений файла по мере обращения к его (файла) данным, что минимизирует число операций проецирования/удаления проекций, выполняемых над файлом при повторном обращении приложения к тем же областям файла.

Кэширование с обратной записью и отложенная запись

Диспетчер кэша реализует кэш с обратной отложенной записью (write-back cache with lazy write). Это означает, что данные, записываемые в файлы, сначала хранятся в страницах кэша в памяти, а потом записываются на диск. Таким образом, записываемые данные в течение некоторого времени накапливаются, после чего сбрасываются на диск пакетом, что уменьшает общее число операций дискового ввода-вывода.

Для сброса страниц кэша диспетчер кэша должен явно вызвать диспетчер памяти, поскольку в ином случае тот записывает на диск содержимое памяти только при нехватке физической памяти. Но, если процесс модифицирует кэшируемые данные, пользователь ожидает, что изменения будут своевременно отражены на диске.

Выбор частоты сброса кэша очень важен. Если слишком часто сбрасывать кэш, быстродействие системы снизится из-за дополнительного ввода-вывода. А при слишком редком сбросе кэша появится риск потери модифицированных файловых данных в случае аварии системы и нехватки физической памяти (которая будет занята чрезмерно большим количеством модифицированных страниц).

Чтобы избежать этих крайностей, раз в секунду в системном рабочем потоке выполняется функция отложенной записи диспетчера кэша, которая сбрасывает на диск (точнее, ставит в очередь на запись) одну восьмую часть измененных страниц системного кэша. Если измененные страницы появляются быстрее, чем сбрасываются, подсистема отложенной записи дополнительно сбрасывает соответствующее дополнительное количество измененных страниц. Реальные операции ввода-вывода выполняются системными рабочими потоками из пула общесистемных критичных рабочих потоков.

ПРИМЕЧАНИЕ Диспетчер кэша предоставляет драйверам файловой системы средства, позволяющие отслеживать, когда и сколько данных было записано в файл. После того как подсистема отложенной записи сбрасывает на диск измененные страницы, диспетчер кэша уведомляет об этом файловую систему, чтобы она обновила свое значение для длины действительных данных файла. (Диспетчер кэша и файловые системы раздельно отслеживают длину действительных данных для файла в памяти.)

Наблюдать за активностью подсистемы отложенной записи позволяют счетчики производительности или системные переменные, перечисленные в таблице 11-11.

Таблица 11-11. *Счетчики и системные переменные, отражающие активность подсистемы отложенной записи*

Объект: счетчик	Системная переменная	Описание
Cache: Lazy Write Flushes/Sec (Кэш: Сбросов ленивой записи/сек)	*CcLazyWriteIos*	Количество сбросов, выполненных подсистемой отложенной записи
Cache: Lazy Write Pages/Sec (Кэш: Страниц 'ленивой' записи/сек)	*CcLazyWritePages*	Количество страниц, записанных подсистемой отложенной записи

Отключение отложенной записи для файла

Если вы создаете временный файл вызовом Windows-функции *CreateFile* с флагом FILE_ATTRIBUTE_TEMPORARY, подсистема отложенной записи не станет записывать измененные страницы этого файла на диск, пока не возникнет существенная нехватка физической памяти или пока файл не будет явно сброшен на диск. Эта особенность подсистемы отложенной записи повышает быстродействие системы: данные, которые в конечном счете могут быть отброшены, на диск сразу не записываются. Приложения обычно удаляют временные файлы вскоре после закрытия.

Принудительное включение в кэше сквозной записи на диск

Поскольку некоторые приложения не терпят ни малейших задержек между записью в файл и реальным обновлением данных на диске, диспетчер кэша поддерживает кэширование со сквозной записью (write-through caching), включаемое для каждого объекта «файл» индивидуально; при этом изменения записываются на диск по мере их внесения. Чтобы включить кэширование со сквозной записью, при вызове функции *CreateFile* надо установить флаг FILE_FLAG_WRITE_THROUGH. В качестве альтернативы поток может явно сбрасывать на диск измененные данные вызовом Windows-функции *FlushFileBuffers*. Вы можете наблюдать за операциями сброса кэша в результате запросов на сквозной ввод-вывод или явных вызовов *FlushFileBuffers* через счетчики производительности или системные переменные, перечисленные в таблице 11-12.

Таблица 11-12. *Счетчики и системные переменные для наблюдения за операциями сброса кэша*

Объект: счетчик	Системная переменная	Описание
Cache: Data Flushes/Sec (Кэш: Сбросов данных/сек)	*CcDataFlushes*	Число сбросов страниц кэша (из-за вызова *FlushFileBuffers* или сквозной записи)
Cache: Data Flush Pages/Sec (Кэш: Страниц сброса данных/сек)	*CcDataPages*	Число страниц кэша, сброшенных явно (вызовом *FlushFileBuffers*) или из-за сквозной записи

Сброс проецируемых файлов

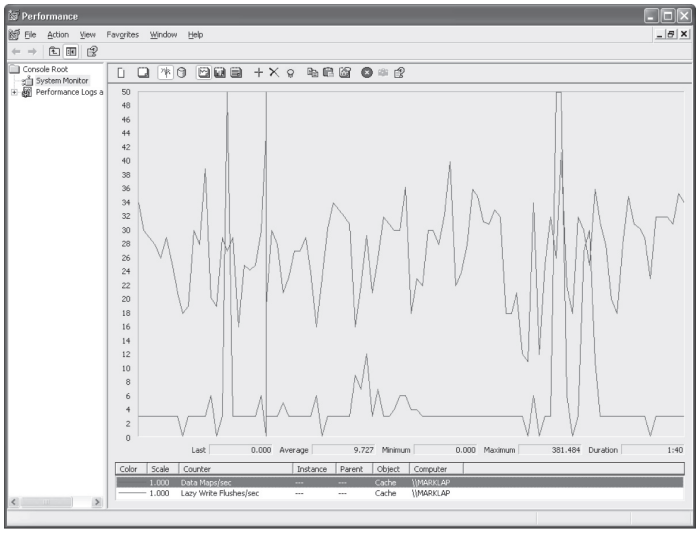
Если подсистема отложенной записи должна записать на диск данные из представления, проецируемого и на адресное пространство другого процесса, ситуация несколько усложняется. Дело в том, что диспетчеру кэша известны лишь страницы, модифицированные им самим. (О страницах, модифицированных другим процессом, знает только этот процесс, так как биты изменения этих страниц находятся в элементах таблиц страниц, принадлежащих исключительно процессу.) Чтобы справиться с этой ситуацией, диспетчер памяти посылает диспетчеру кэша соответствующее уведомление в тот момент, когда пользователь проецирует какой-либо файл. При сбросе такого файла из кэша (например, в результате вызова Windows-функции *FlushFileBuffers*) диспетчер кэша записывает на диск модифицированные страницы из кэша, а затем проверяет, не спроецирован ли этот файл другим процессом. Если да, диспетчер кэша сбрасывает все представление раздела для того, чтобы записать любые страницы, которые мог модифицировать второй процесс. Если пользователь заканчивает проецировать представление файла, открытого и в кэше, модифицированные страницы помечаются как измененные, чтобы при последующем сбросе представления подсистемой отложенной записи эти страницы были записаны на диск. Такой алгоритм действует всякий раз, когда возникает следующая последовательность событий.

1. Пользователь удаляет проекцию представления.
2. Процесс сбрасывает файловые буферы.

При ином порядке событий предсказать, какие страницы будут записаны на диск, нельзя.

ЭКСПЕРИМЕНТ: наблюдение за операциями сброса кэша

Вы можете увидеть, как диспетчер кэша проецирует представления в системный кэш и сбрасывает страницы на диск, запустив оснастку Performance (Производительность) и добавив счетчики Data Maps/sec (Отображений данных/сек) и Lazy Write Flushes/sec (Сбросов ленивой записи/сек), а затем скопировав большой файл из одного места в другое. На следующей иллюстрации показаны графики, относящиеся к Data Maps/sec (верхний) и к Lazy Write Flushes/sec (нижний).



Дросселирование записи

Файловая система и диспетчер кэша должны определять, повлияет ли запрос кэшированной записи на производительность системы, и исходя из этого планировать отложенные операции записи. Сначала файловая система через функцию *CcCanIWrite* выясняет у диспетчера кэша, можно ли записать определенное число байтов прямо сейчас без ущерба для производительности, и при необходимости блокирует запись. Далее она настраивает обратный вызов диспетчера кэша для автоматической записи данных на диск, когда запись вновь будет разрешена вызовом *CcDeferWrite*. Получив уведомление о предстоящей операции записи, диспетчер кэша определяет, сколько измененных страниц находится в кэше и какой объем физической памяти доступен. Если свободных физических страниц мало, диспетчер кэша немедленно блокирует поток файловой системы, выдавший запрос на запись данных в кэш. Подсистема отложенной записи сбрасывает часть измененных страниц на диск, после чего разрешает продолжить выполнение блокированного потока файловой системы. Такой механизм, называемый *дросселированием записи* (write throttling), предотвращает падение быстродействия системы из-за нехватки памяти при операциях записи большого объема данных, инициируемых файловой системой или сетевым сервером.

ПРИМЕЧАНИЕ Дросселирование записи оказывает глобальное влияние на систему, так как ресурс, на котором основан этот механизм, — свободная физическая память — является глобальным. И если интенсивная запись на медленное устройство вызывает дросселирование, это распространяется на операции записи и на другие устройства.

Пороговое число измененных страниц (dirty page threshold) — это количество страниц, хранимых системным кэшем в памяти, по достижении которого пробуждается поток подсистемы отложенной записи для сброса страниц на диск. Это значение вычисляется при инициализации системы и зависит от объема физической памяти и параметра реестра LargeSystemCache, как мы уже объясняли.

Алгоритм расчета порогового числа измененных страниц представлен в таблице 11-13. Результат расчета с использованием этого алгоритма игнорируется, если максимальный размер системного рабочего набора превышает 4 Мб, — а именно так зачастую и происходит. (Определение малого, среднего и большого объема системной памяти см. в главе 7.) Когда максимальный размер рабочего набора превышает 4 Мб, пороговое число измененных страниц устанавливается равным максимальному размеру системного рабочего набора за вычетом 2 Мб.

Таблица 11-13. *Алгоритм расчета порогового числа измененных страниц*

Объем системной памяти	Пороговое число измененных страниц
Малый	Число физических страниц / 8
Средний	Число физических страниц / 4
Большой	Сумма двух вышеуказанных значений

Дросселирование записи также полезно для сетевых редиректоров, передающих данные по медленным коммуникационным каналам. Вообразите, например, локальный процесс, записывающий большой объем данных в удаленную файловую систему по каналу, работающему со скоростью 9600 бод. Данные не попадут на удаленный диск, пока подсистема отложенной записи диспетчера кэша не сбросит кэш на диск. Если редиректор накапливает много измененных страниц, одновременно сбрасываемых на диск, на принимающей стороне может истечь время ожидания данных до окончания их передачи. Развитие событий по такому сценарию можно предотвратить с помощью функции диспетчера кэша *CcSetDirtyPageThreshold*, позволяющей сетевым редиректорам устанавливать лимит на количество измененных страниц, которые можно накапливать без сброса на диск. Ограничивая число измененных страниц, редиректор гарантирует, что операции сброса кэша не вызовут таймаута.

ПРИМЕЧАНИЕ В Windows XP и выше сетевые редиректоры не задают пороговое значение измененных страниц, вместо этого полагаясь на системные значения по умолчанию.

ЭКСПЕРИМЕНТ: просмотр параметров дросселирования записи

Команда *!defwrites* отладчика ядра выводит дамп значений переменных ядра, используемых диспетчером кэша для определения момента дросселирования операций записи.

```
kd> !defwrites
*** Cache Write Throttle Analysis ***

        CcTotalDirtyPages:                 758 (    3032 Kb)
        CcDirtyPageThreshold:              770 (    3080 Kb)
        MmAvailablePages:                42255 (  169020 Kb)
        MmThrottleTop:                     250 (    1000 Kb)
        MmThrottleBottom:                   30 (     120 Kb)
        MmModifiedPageListHead.Total:      689 (    2756 Kb)

CcTotalDirtyPages within 64 (max charge) pages of the threshold,
  writes may be throttled

Check critical workqueue for the lazy writer, !exqueue 16
```

Как видите, число измененных страниц близко к пороговому значению, при котором начинается дросселирование записи (*CcDirtyPageThreshold*), и поэтому, если бы в ходе эксперимента процесс попытался записать более 12 страниц (48 Кб), эти операции были бы отложены до тех пор, пока подсистема отложенной записи не уменьшила бы число измененных страниц.

Системные потоки

Как уже говорилось, диспетчер кэша выполняет отложенную запись и опережающее чтение, передавая запросы в общий пул критичных системных рабочих потоков. Однако в системах с малым и средним объемом памяти он использует на один поток меньше общего количества критичных системных рабочих потоков, а в системах с большим объемом памяти — на два.

Внутренне диспетчер кэша организует свои запросы в два списка (которые все равно обслуживаются одним и тем же набором рабочих потоков исполнительной системы):

- *экспресс-очередь* (express queue) — для операций опережающего чтения;
- *регулярная очередь* (regular queue) — для отложенной записи измененных данных, подлежащих сбросу, обратной записи и отложенного закрытия файлов.

Чтобы отслеживать рабочие элементы, направленные рабочим потокам, диспетчер кэша создает собственный ассоциативный список (индивидуальный для каждого процессора и имеющий фиксированную длину) структур рабочих элементов, поставленных в очереди рабочих потоков (ассоциативные списки обсуждаются в главе 7). Число элементов в очереди рабочего потока определяется объемом системной памяти и для систем Windows Professional составляет 32, 64 или 128 в системах с малым, средним или большим объемом памяти соответственно (для систем Windows Server с большим объемом памяти — 256).

Резюме

Диспетчер кэша предоставляет быстродействующий интеллектуальный механизм для уменьшения интенсивности дискового ввода-вывода и увеличения общей пропускной способности системы. Осуществляя кэширование на основе виртуальных блоков, диспетчер кэша может выполнять интеллектуальное опережающее чтение. Используя для обращения к файловым данным механизм проецирования файлов, диспетчер кэша предоставляет специальный механизм для быстрого ввода-вывода, который уменьшает нагрузку на процессор при выполнении операций чтения и записи и позволяет возложить все управление физической памятью на диспетчер памяти. А это избавляет от дублирования кода и повышает эффективность работы операционной системы.

ГЛАВА 12

Файловые системы

В начале этой главы мы даем обзор файловых систем, поддерживаемых Windows, а также описываем типы драйверов файловых систем и принципы их работы, в том числе способы взаимодействия с другими компонентами операционной системы, например с диспетчерами памяти и кэша. Затем поясняем, как пользоваться утилитой Filemon (*www.sysinternals.com*) для анализа проблем, связанных с доступом к файловой системе. Мы рассмотрим «родной» для Windows формат файловой системы NTFS и особенности этой файловой системы — сжатие данных, способность к восстановлению, поддержку квот и шифрование.

Чтобы полностью усвоить материал этой главы, вы должны понимать терминологию, введенную в главе 10, в том числе термины «том» и «раздел». Кроме того, вам должны быть знакомы следующие дополнительные термины.

- *Секторы* — аппаратно адресуемые блоки носителя. Размер секторов на жестких дисках в x86-системах почти всегда равен 512 байтам. Таким образом, если операционная система должна модифицировать 632-й байт диска, она записывает 512-байтовый блок данных во второй сектор диска.
- *Форматы файловых систем* определяют принципы хранения данных на носителе и влияют на характеристики файловой системы. Например, файловая система, формат которой не допускает сопоставления прав доступа с файлами и каталогами, не поддерживает защиту. Формат файловой системы также может налагать ограничения на размеры файлов и емкости поддерживаемых устройств внешней памяти. Наконец, некоторые форматы файловых систем эффективно реализуют поддержку либо больших, либо малых файлов и дисков.
- *Кластеры* — адресуемые блоки, используемые многими файловыми системами. Размер кластера всегда кратен размеру сектора (рис. 12-1). Файловая система использует кластеры для более эффективного управления дисковым пространством: кластеры, размер которых превышает размер сектора, позволяют разбить диск на блоки меньшей длины — управлять такими блоками легче, чем секторами. Потенциальный недостаток кластеров большего размера — менее эффективное использование дискового пространства, или внутренняя фрагментация, которая возникает из-за того, что размеры файлов редко бывают кратны размеру кластера.

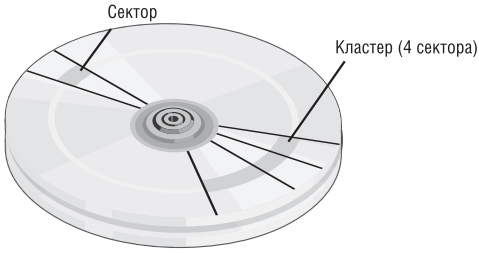


Рис. 12-1. *Секторы и кластер на диске*

- *Метаданные* — это данные, хранящиеся на томе и необходимые для поддержки управления файловой системой. Как правило, они недоступны приложениям. Метаданные включают, например, информацию, определяющую местонахождение файлов и каталогов на томе.

Файловые системы Windows

Windows поддерживает файловые системы:

- CDFS;
- UDF;
- FAT12, FAT16 и FAT32;
- NTFS.

Как вы еще увидите, каждая из этих файловых систем оптимальна для определенной среды.

CDFS

CDFS, или файловая система CD-ROM (только для чтения), обслуживается драйвером \Windows\System32\Drivers\Cdfs.sys, который поддерживает надмножества форматов ISO-9660 и Joliet. Если формат ISO-9660 сравнительно прост и имеет ряд ограничений, например имена с буквами верхнего регистра в кодировке ASCII и максимальной длиной в 32 символа, то формат Joliet более гибок и поддерживает Unicode-имена произвольной длины. Если на диске присутствуют структуры для обоих форматов (чтобы обеспечить максимальную совместимость), CDFS использует формат Joliet. CDFS присущ ряд ограничений:

- максимальная длина файлов не должна превышать 4 Гб;
- число каталогов не может превышать 65 535.

CDFS считается унаследованным форматом, поскольку индустрия уже приняла в качестве стандарта для носителей, предназначенных только для чтения, формат Universal Disk Format (UDF).

UDF

Файловая система UDF в Windows является UDF-совместимой реализацией OSTA (Optical Storage Technology Association). (UDF является подмножеством

формата ISO-13346 с расширениями для поддержки CD-R, DVD-R/RW и т. д.) OSTA определила UDF в 1995 году как формат магнитооптических носителей, главным образом DVD-ROM, предназначенный для замены формата ISO-9660. UDF включен в спецификацию DVD и более гибок, чем CDFS. Драйвер UDF поддерживает UDF версий 1.02 и 1.5 в Windows 2000, а также версий 2.0 и 2.01 в Windows XP и Windows Server 2003. Файловые системы UDF обладают следующими преимуществами:

- длина имен файлов и каталогов может быть до 254 символов в ASCII-кодировке или до 127 символов в Unicode-кодировке;
- файлы могут быть разреженными (sparse);
- размеры файлов задаются 64-битными значениями.

Хотя формат UDF разрабатывался с учетом особенностей перезаписываемых носителей, драйвер UDF в Windows (\Windows\System32\Drivers\Udfs.sys) поддерживает носители только для чтения. Кроме того, в Windows не реализована поддержка других возможностей UDF, в частности именованных потоков, списков управления доступом и расширенных атрибутов.

FAT12, FAT16 и FAT32

Windows поддерживает файловую систему FAT по трем причинам: для возможности обновления операционной системы с прежних версий Windows до современных, для совместимости с другими операционными системами при многовариантной загрузке и как формат гибких дисков. Драйвер файловой системы FAT в Windows реализован в \Windows\System32\Drivers\Fastfat.sys.

В название каждого формата FAT входит число, которое указывает разрядность, применяемую для идентификации кластеров на диске. 12-разрядный идентификатор кластеров в FAT12 ограничивает размер дискового раздела 2^{12} (4096) кластерами. В Windows используются кластеры размером от 512 байтов до 8 Кб, так что размер тома FAT12 ограничен 32 Мб. Поэтому Windows использует FAT12 как формат 5- и 3,5-дюймовых дискет, способных хранить до 1,44 Мб данных.

> **ПРИМЕЧАНИЕ** Все файловые системы FAT резервируют на томе первые два кластера и последние 16, так что число доступных для использования кластеров на томе, например FAT12, чуть меньше 4096.

FAT16 — за счет 16-разрядных идентификаторов кластеров — может адресовать до 2^{16} (65 536) кластеров. В Windows размер кластера FAT16 варьируется от 512 байтов до 64 Кб, поэтому размер FAT16-тома ограничен 4 Гб. Размер кластеров, используемых Windows, зависит от размера тома (таблица 12-1). Если вы форматируете том размером менее 16 Мб для FAT с помощью команды *format* или оснастки Disk Management (Управление дисками), Windows вместо FAT16 использует FAT12.

Таблица 12-1. *Размеры кластеров в FAT16 в Windows по умолчанию*

Размер тома (Мб)	Размер кластера
0–32	512 байтов
33–64	1 Кб
65–128	2 Кб
129–256	4 Кб
257–511	8 Кб
512–1023	16 Кб
1024–2047	32 Кб
2048–4095	64 Кб

Том FAT делится на несколько областей (рис. 12-2). Таблица размещения файлов (file allocation table, FAT), от которой и произошло название файловой системы FAT, имеет по одной записи для каждого кластера тома. Поскольку таблица размещения файлов критична для успешной интерпретации содержимого тома, FAT поддерживает две копии этой таблицы. Так что, если драйвер файловой системы или программа проверки целостности диска (вроде Chkdsk) не сумеет получить доступ к одной из копий FAT (например, из-за плохого сектора на диске), она сможет использовать вторую копию.

Рис. 12-2. *Организация FAT*

Записи в таблице FAT определяют цепочки размещения файлов и каталогов (рис. 12-3), где отдельные звенья представляют собой указатели на следующий кластер с данными файла. Элемент каталога для файла хранит начальный кластер файла. Последний элемент цепочки размещения файла содержит зарезервированное значение 0xFFFF для FAT16 и 0xFFF для FAT12. Записи FAT, описывающие свободные кластеры, содержат нулевые значения. На рис. 12-3 показан файл FILE1, которому назначены кластеры 2, 3 и 4; FILE2 фрагментирован и использует кластеры 5, 6 и 8; а FILE3 занимает только кластер 7. Чтение файла с FAT-тома может потребовать просмотра больших блоков таблицы размещения файлов для поиска всех его цепочек размещения.

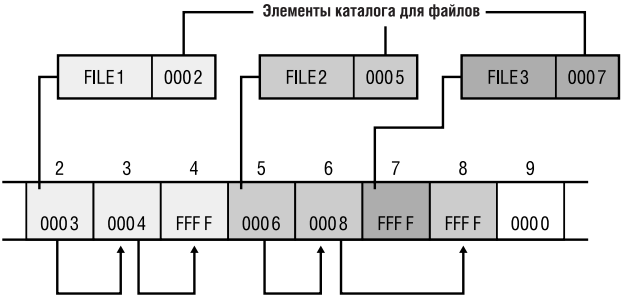


Рис. 12-3. *Пример цепочек размещения файлов в FAT*

В начале тома FAT12 или FAT16 заранее выделяется место для корневого каталога, достаточное для хранения 256 записей (элементов), что ограничивает число файлов и каталогов в корневом каталоге (в FAT32 такого ограничения нет). Элемент каталога FAT, размер которого составляет 32 байта, хранит имя файла, его размер, начальный кластер и метку времени (время создания, последнего доступа и т. д.). Если имя файла состоит из Unicode-символов или не соответствует правилам именования по формуле «8.3», принятым в MS-DOS, оно считается длинным и для его хранения выделяются дополнительные элементы каталога. Вспомогательные элементы предшествуют главному элементу для файла. На рис. 12-4 показан пример элемента каталога для файла с именем «The quick brown fox». Система создала представление этого имени в формате «8.3», THEQUI~1.FOX (в элементе каталога вы не увидите «.», поскольку предполагается, что точка следует после восьмого символа), и использовала два дополнительных элемента для хранения длинного Unicode-имени. Каждая строка на рис. 12-4 состоит из 16 байтов.

Рис. 12-4. *Элементы каталога FAT*

FAT32 — более новая файловая система на основе формата FAT; она поддерживается Windows 95 OSR2, Windows 98 и Windows Millennium Edition. FAT32 использует 32-разрядные идентификаторы кластеров, но при этом резервирует старшие 4 бита, так что эффективный размер идентификатора кластера составляет 28 бит. Поскольку максимальный размер кластеров FAT32 равен 32 Кб, теоретически FAT32 может работать с 8-терабайтными томами. Windows ограничивает размер новых томов FAT32 до 32 Гб, хотя поддерживает существующие тома FAT32 большего размера (созданные в других операционных системах). Большее число кластеров, поддерживаемое FAT32, позволяет ей управлять дисками более эффективно, чем FAT16. FAT32 может использовать 512-байтовые кластеры для томов размером до 128 Мб. Размеры кластеров на томах FAT32 по умолчанию показаны в таблице 12-2.

Таблица 12-2. *Размер кластеров на томах FAT32 по умолчанию*

Размер раздела	Размер кластера (Кб)
От 32 Мб до 8 Гб	4
8–16 Гб	8
16–32 Гб	16
32 Гб	32

Помимо большего предельного числа кластеров преимуществом FAT32 перед FAT12 и FAT16 является тот факт, что место хранения корневого каталога FAT32 не ограничено предопределенной областью тома, поэтому его размер не ограничен. Кроме того, для большей надежности FAT32 хранит вторую копию загрузочного сектора*. В FAT32, как и в FAT16, максимальный размер файла равен 4 Гб, поскольку длина файла в каталоге описывается 32-битным числом.

ПРИМЕЧАНИЕ В Windows XP введена поддержка FAT32 на устройствах DVD-RAM.

NTFS

Как мы уже говорили в начале главы, NTFS — встроенная («родная») файловая система Windows. NTFS использует 64-разрядные номера кластеров. Это позволяет NTFS адресовать тома размером до 16 экзабайт (16 миллиардов Гб). Однако Windows ограничивает размеры томов NTFS до значений, при которых возможна адресация 32-разрядными кластерами, т. е. до 128 Тб (с использованием кластеров по 64 Кб). В таблице 12-3 перечислены размеры кластеров на томах NTFS по умолчанию (эти значения можно изменить при форматировании тома NTFS).

Таблица 12-3. *Размеры кластеров на томах NTFS по умолчанию*

Размер тома	Размер кластера
512 Мб и менее	512 байтов
513–1024 Мб	1 Кб
1025–2048 Мб	2 Кб
Более 2048 Мб	4 Кб

NTFS поддерживает ряд дополнительных возможностей — защиту файлов и каталогов, дисковые квоты, сжатие файлов, символьные ссылки на основе каталогов и шифрование. Одно из важнейших свойств NTFS — *восстанавливаемость*. При неожиданной остановке системы целостность метаданных тома FAT может быть утрачена, что вызовет повреждение структуры каталогов и значительного объема данных. NTFS ведет журнал изменений метаданных путем протоколирования транзакций, поэтому целостность структур файловой системы может быть восстановлена без потери информации о структуре файлов или каталогов. (Однако данные файлов могут быть потеряны.)

* Точнее — загрузочной записи, которая включает несколько секторов. Подробную информацию о FAT32 см. в книге «Ресурсы Microsoft Windows 98». — *Прим. перев.*

Подробнее о структурах данных и дополнительных возможностях NTFS мы поговорим позже.

Архитектура драйвера файловой системы

Драйвер файловой системы (file system driver, FSD) управляет форматом файловой системы. Хотя FSD выполняются в режиме ядра, у них есть целый ряд особенностей по сравнению со стандартными драйверами режима ядра. Возможно, самой важной особенностью является то, что они должны регистрироваться у диспетчера ввода-вывода и более интенсивно взаимодействовать с ним. Кроме того, для большей производительности FSD обычно полагаются на сервисы диспетчера кэша. Таким образом, FSD используют более широкий набор функций, экспортируемых Ntoskrnl, чем стандартные драйверы. Если для создания стандартных драйверов режима ядра требуется Windows DDK, то для создания драйверов файловых систем понадобится Windows Installable File System (IFS) Kit (подробнее о DDK см. главу 1; подробнее о IFS Kit см. *www.microsoft.com/whdc/devtools/ifskit*).

В Windows два типа драйверов файловых систем:

- *локальные FSD*, управляющие дисковыми томами, подключенными непосредственно к компьютеру;
- *сетевые FSD*, позволяющие обращаться к дисковым томам, подключенным к удаленным компьютерам.

Локальные FSD

К локальным FSD относятся Ntfs.sys, Fastfat.sys, Udfs.sys, Cdfs.sys и Raw FSD (интегрированный в Ntoskrnl.exe). На рис. 12-5 показана упрощенная схема взаимодействия локальных FSD с диспетчером ввода-вывода и драйверами устройств внешней памяти. Как мы поясняли в разделе «Монтирование томов» главы 10, локальный FSD должен зарегистрироваться у диспетчера ввода-вывода. После регистрации FSD диспетчер ввода-вывода может вызывать его для распознавания томов при первом обращении к ним системы или одного из приложений. Процесс распознавания включает анализ загрузочного сектора тома и, как правило, метаданных файловой системы для проверки ее целостности.

Все поддерживаемые Windows файловые системы резервируют первый сектор тома как загрузочный. Загрузочный сектор содержит достаточно информации, чтобы FSD мог идентифицировать свой формат файловой системы тома и найти любые метаданные, хранящиеся на этом томе.

Распознав том, FSD создает объект «устройство», представляющий смонтированную файловую систему. Диспетчер ввода-вывода связывает объект «устройство» тома, созданный драйвером устройства внешней памяти (далее — объект тома), с объектом «устройство», созданным FSD (далее — объект FSD), через блок параметров тома (VPB). Это приводит к тому, что диспетчер ввода-вывода перенаправляет через VPB запросы ввода-вывода, адресованные объекту тома, на объект FSD (подробнее о VPB см. главу 10).

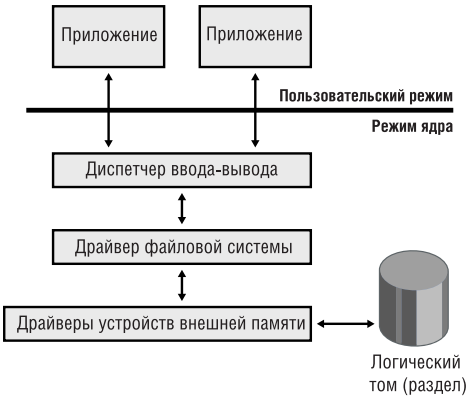


Рис. 12-5. *Локальный FSD*

Для большей производительности локальные FSD обычно используют диспетчер кэша, который кэширует данные файловой системы, в том числе ее метаданные. Они также интегрируются с диспетчером памяти, что позволяет корректно реализовать проецирование файлов. Например, всякий раз, когда приложение пытается обрезать файл, они должны запрашивать диспетчер памяти, чтобы убедиться, что за точкой отсечения файл не проецируется ни одним процессом. Windows не разрешает удалять данные файла, проецируемого приложением.

Локальные FSD также поддерживают операции демонтирования файловой системы, позволяющие операционной системе отсоединять FSD от объекта тома. Демонтирование происходит каждый раз, когда приложение напрямую обращается к содержимому тома или когда происходит смена носителя, сопоставленного с томом. При первом обращении приложения к носителю после демонтирования диспетчер ввода-вывода повторно инициирует операцию монтирования тома для этого носителя.

Удаленные FSD

Удаленные FSD состоят из двух компонентов: клиента и сервера. Удаленный FSD на клиентской стороне позволяет приложениям обращаться к удаленным файлам и каталогам. Клиентский FSD принимает запросы ввода-вывода от приложений и транслирует их в команды протокола сетевой файловой системы, посылаемые через сеть компоненту на серверной стороне, которым обычно является удаленный FSD. Серверный FSD принимает команды, поступающие по сетевому соединению, и выполняет их. При этом он выдает запросы на ввод-вывод локальному FSD, управляющему томом, на котором расположен нужный файл или каталог.

Windows включает клиентский удаленный FSD, LANMan Redirector (редиректор), и серверный удаленный FSD, LANMan Server (сервер) (\Windows\System32\Drivers\Srv.sys). Редиректор реализован в виде комбинации порт- и минипорт-драйверов, где порт-драйвер (\Windows\System32\Drivers \Rdbss.sys) представляет собой библиотеку подпрограмм, а минипорт-драйвер (\Win-

dows\System32\Drivers\Mrxsmb.sys) использует сервисы, реализуемые порт-драйвером. Еще один минипорт-драйвер редиректора — WebDAV (\Windows\System32\Drivers\Mrxdav.sys), который реализует клиентскую часть поддержки доступа к файлам по HTTP. Модель «порт-минипорт» упрощает разработку редиректора, потому что порт-драйвер, совместно используемый всеми минипорт-драйверами удаленных FSD, берет на себя многие рутинные операции, требуемые при взаимодействии между клиентским FSD и диспетчером ввода-вывода Windows. В дополнение к FSD-компонентам LANMan Redirector и LANMan Server включают Windows-службы рабочей станции и сервера соответственно. Взаимодействие между клиентом и сервером при доступе к файлам на серверной стороне через редиректор и серверные FSD показано на рис. 12-6.

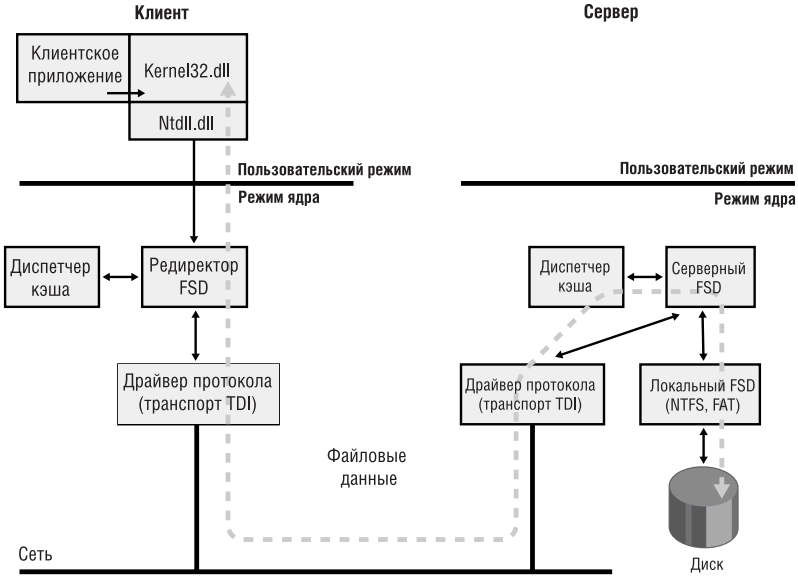


Рис. 12-6. *Обмен файлами по протоколу CIFS*

Для форматирования сообщений, которыми обмениваются редиректор и сервер, Windows использует протокол CIFS (Common Internet File System). CIFS — это версия протокола Microsoft SMB (Server Message Block). (Подробнее о CIFS см. книгу «Сети TCP/IP. Ресурсы Microsoft Windows 2000 Server» и сайт *www.cifs.com*.)

Как и локальные FSD, удаленные FSD на клиентской стороне обычно используют сервисы диспетчера кэша для локального кэширования файловых данных, относящихся к удаленным файлам и каталогам. Однако удаленный FSD на клиентской стороне должен реализовать протокол поддержки когерентности распределенного кэша, называемый *oplock* (opportunistic locking), гарантирующий, что любое приложение при обращении к удаленному файлу получит те же данные, что и приложения на других компьютерах в сети. Хотя удаленные FSD на серверной стороне участвуют в поддержании коге-

рентности клиентских кэшей, они не кэшируют данные локальных FSD, поскольку те сами кэшируют свои данные.

Когда клиент пытается обратиться к файлу на сервере, он должен сначала запросить oplock. Вид доступного клиенту кэширования, определяется типом oplock, предоставляемого сервером. Существует три основных типа oplock.

- Level I oplock предоставляется при монопольном доступе клиента к файлу. Клиент, удерживающий для файла oplock этого типа, может кэшировать операции как чтения, так и записи.
- Level II oplock — разделяемая блокировка файла. Клиенты, удерживающие oplock этого типа, могут кэшировать операции чтения, но запись в файл делает Level II oplock недействительным.
- Batch oplock — самый либеральный тип oplock. Он позволяет клиенту не только читать и записывать файл, но и открывать и закрывать его, не запрашивая дополнительные oplock. Batch oplock, как правило, используется только для поддержки выполнения пакетных (командных) файлов, которые могут неоднократно закрываться и открываться в процессе выполнения.

В отсутствие oplock клиент не может осуществлять локальное кэширование ни операций чтения, ни операций записи; вместо этого он должен получать данные с сервера и посылать все изменения непосредственно на сервер.

Проиллюстрировать работу oplock поможет пример на рис. 12-7. Первому клиенту, открывающему файл, сервер автоматически предоставляет Level I oplock. Редиректор на клиентской стороне кэширует файловые данные при чтении и записи в кэше файловой системы локальной машины. Если тот же файл открывает второй клиент, он также запрашивает Level I oplock. Теперь уже два клиента обращаются к одному и тому же файлу, поэтому сервер должен принять меры для согласования представления данных файла обоим клиентам. Если первый клиент произвел запись в файл (этот случай и показан на рис. 12-7), сервер отзывает oplock и больше не предоставляет его ни одному клиенту. После отзыва oplock первый клиент сбрасывает все кэшированные данные файла обратно на сервер.

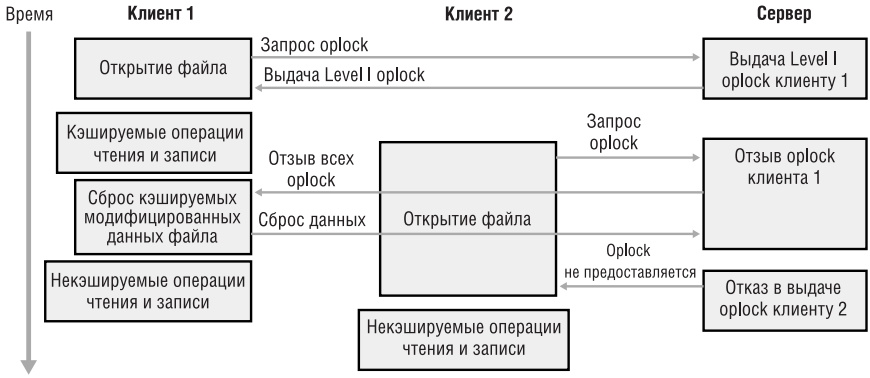
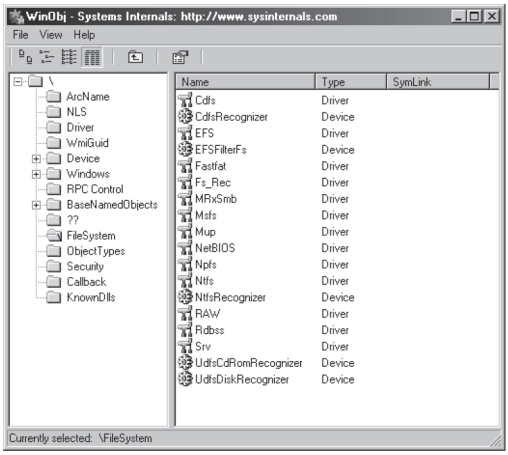


Рис. 12-7. *Пример работы oplock*

Если бы первый клиент не произвел запись, его oplock был бы понижен до Level II oplock, т. е. до oplock того же типа, который сервер предоставляет второму клиенту. В этом случае оба клиента могли бы кэшировать операции чтения, но после операции записи любым из клиентов сервер отозвал бы их oplock, и последующие операции были бы некэшируемыми. Однажды отозванный, oplock больше не предоставляется для этого экземпляра открытого файла. Однако, если клиент закрывает файл и повторно открывает его, сервер заново решает, какой oplock следует предоставить клиенту. Решение сервера зависит от того, открыт ли файл другими клиентами и производил ли хоть один из них запись в этот файл.

ЭКСПЕРИМЕНТ: просмотр списка зарегистрированных файловых систем

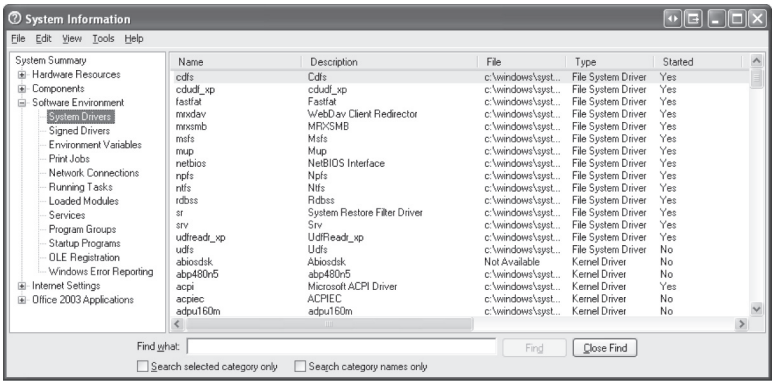
Диспетчер ввода-вывода, загружая в память драйвер устройства, обычно присваивает имя объекту «драйвер», который создается для представления этого драйвера. Этот объект помещается в каталог \Drivers диспетчера объектов. Объекты «драйвер» любого загружаемого диспетчером ввода-вывода драйвера с атрибутом Type, равным SERVICE_FILE_SYSTEM_DRIVER (2), помещаются этим диспетчером в каталог \FileSystem. Таким образом, утилита типа Winobj (*www.sysinternals.com*) позволяет увидеть зарегистрированные файловые системы, как показано на следующей иллюстрации. (Заметьте, что некоторые драйверы файловых систем помещают в каталог \FileSystem и объекты «устройство».)



Еще один способ увидеть зарегистрированные файловые системы — запустить программу System Information (Сведения о системе). В Windows 2000 запустите оснастку Computer Management (Управление компьютером) и выберите Drivers (Драйверы) в Software Environment (Программная среда) в узле System Information (Сведения о системе); в Windows XP и Windows Server 2003 запустите Msinfo32 и выберите System Drivers (Системные драйверы) в узле Software Environ-

см. след. стр.

ment (Программная среда). Отсортируйте список драйверов, щелкнув колонку Type (Тип), и драйверы с атрибутом SERVICE_FILE_SYSTEM_DRIVER окажутся в начале списка.



Заметьте: если драйвер регистрируется как SERVICE_FILE_SYSTEM_DRIVER, это еще не означает, что он служит локальным или удаленным FSD. Например, Npfs (Named Pipe File System), присутствующий на только что показанной иллюстрации, на самом деле является драйвером сетевого API — он поддерживает именованные каналы, но реализует закрытое пространство имен и поэтому в какой-то мере подобен драйверу файловой системы. Пространство имен Npfs исследуется в одном из экспериментов в главе 13.

Работа файловой системы

Система и приложения могут обращаться к файлам двумя способами: напрямую (через функции ввода-вывода вроде *ReadFile* и *WriteFile*) и косвенно, путем чтения или записи части своего адресного пространства, где находится раздел проецируемого файла (подробнее о проецируемых файлах см. главу 7). Упрощенная схема на рис. 12-8 иллюстрирует компоненты, участвующие в работе файловой системы, и способы их взаимодействия. Как видите, есть несколько путей вызова FSD:

- из пользовательского или системного потока, выполняющего явную операцию файлового ввода-вывода;
- из подсистем записи модифицированных и спроецированных страниц, принадлежащих диспетчеру памяти;
- неявно из подсистемы отложенной записи, принадлежащей диспетчеру кэша;
- неявно из потока опережающего чтения, принадлежащего диспетчеру кэша;
- из обработчика ошибок страниц, принадлежащего диспетчеру памяти.

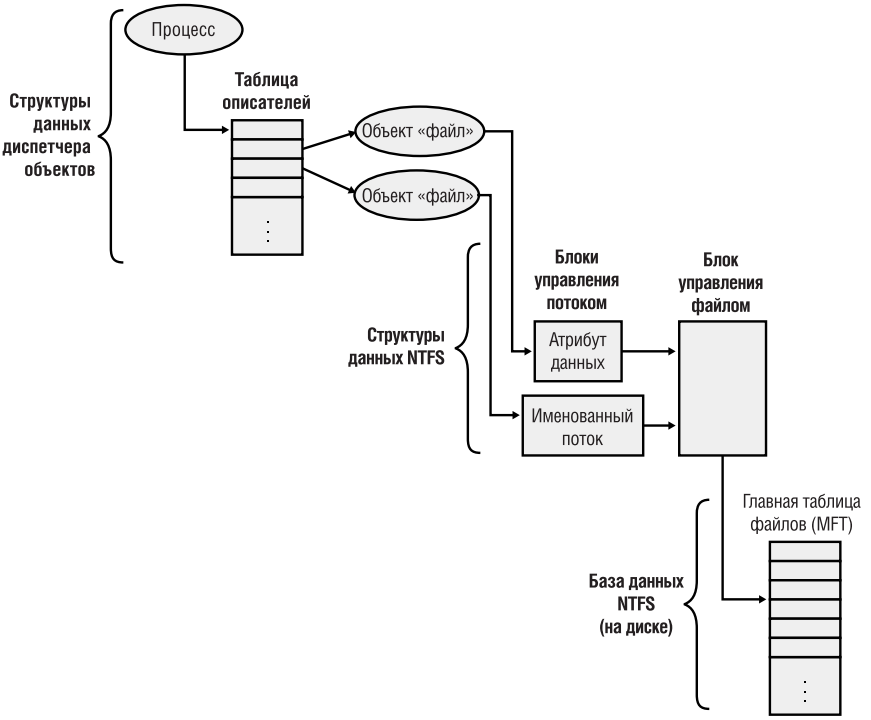


Рис. 12-8. *Компоненты, участвующие в операциях ввода-вывода файловой системы*

Явный файловый ввод-вывод

Наиболее очевидный способ доступа приложения к файлам — вызов Windows-функций ввода-вывода, например *CreateFile*, *ReadFile* и *WriteFile*. Приложение открывает файл с помощью *CreateFile*, а затем читает, записывает и удаляет его, передавая описатель файла, возвращенный *CreateFile*, другим Windows-функциям. *CreateFile*, реализованная в Kernel32.dll, вызывает встроенную функцию *NtCreateFile* и формирует полное имя файла, обрабатывая символы «.» и «..» и предваряя путь строкой «\??» (например, \??\C:\Daryl \Todo.txt).

Чтобы открыть файл, системный сервис *NtCreateFile* вызывает функцию *ObOpenObjectByName*, которая выполняет разбор имени, начиная с корневого каталога диспетчера объектов и первого компонента полного имени («??»). В главе 3 дано подробное описание разрешения имен диспетчером объектов, а здесь мы поясним, как происходит поиск букв диска для томов.

Первое, что делает диспетчер объектов, — транслирует \?? в каталог пространства имен, индивидуальный для сеанса, в котором выполняется данный процесс; на этот каталог ссылается поле *DosDevicesDirectory* в структуре карты устройств (device map structure) в объекте «процесс». В системах Windows 2000 без Terminal Services поле *DosDevicesDirectory* ссылается на каталог \??, а в системах Windows 2000 без Terminal Services карта устройств ссылается

на индивидуальный для каждого сеанса каталог, где хранятся объекты «символьная ссылка», представляющие все действительные буквы дисков для томов. Однако в Windows XP и Windows Server 2003 в таком каталоге обычно содержатся лишь имена томов для общих сетевых ресурсов, поэтому в этих ОС диспетчер объектов, не найдя имя (в данном примере — C:) в индивидуальном для сеанса каталоге, переходит к поиску в каталоге, на который ссылается поле *GlobalDosDevicesDirectory* карты устройств, сопоставленной с индивидуальным для сеанса каталогом. *GlobalDosDevicesDirectory* всегда указывает на каталог \Global??, где Windows XP и Windows Server 2003 хранят буквы дисков для локальных томов. (Подробнее о пространстве имен сеанса см. одноименный раздел в главе 3.)

Символьная ссылка для буквы диска, присвоенной тому, указывает на объект тома в каталоге \Device, поэтому диспетчер объектов, распознав объект тома, передает остаток строки с именем в функцию *IopParseDevice*, зарегистрированную диспетчером ввода-вывода для объектов «устройство». (На томах динамических дисков символьная ссылка указывает на промежуточную ссылку, которая в свою очередь указывает на объект тома.) На рис. 12-9 показано, как происходит доступ к объектам томов через пространство имен диспетчера объектов. В данном случае (система Windows 2000 без Terminal Services) символьная ссылка \??\C: указывает на объект тома \Device\HarddiskVolume1.

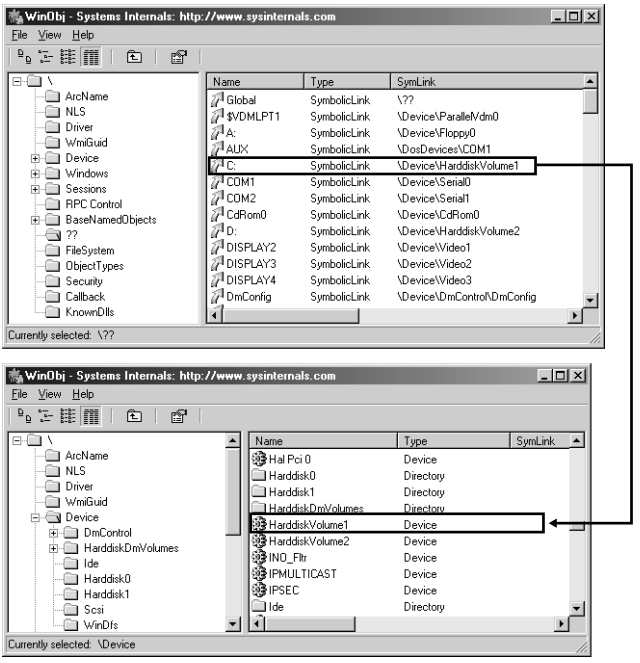


Рис. 12-9. *Разрешение букв дисков*

Заблокировав контекст защиты вызывающего потока и получив информацию о защите из его маркера, *IopParseDevice* генерирует пакет запроса

ввода-вывода (IRP) типа IRP_MJ_CREATE, создает объект «файл», в котором запоминается имя открываемого файла, и по ссылке в VPB объекта тома находит объект «устройство» смонтированной файловой системы тома. Далее, используя *IoCallDriver*, она передает IRP драйверу файловой системы, которому принадлежит данный объект «устройство».

Когда FSD получает IRP типа IRP_MJ_CREATE, он ищет указанный файл, проверяет права доступа и, если файл есть и пользователь имеет права на запрошенный вид доступа к файлу, возвращает код успешного завершения. Диспетчер объектов создает в таблице описателей, принадлежащей процессу, описатель объекта «файл», который передается назад по цепочке вызовов, в конечном счете достигая приложения в виде параметра, возвращаемого *CreateFile*. Если файловой системе не удается создать файл, диспетчер ввода-вывода удаляет созданный для него объект «файл».

Мы опустили здесь детали, относящиеся к тому, как FSD находит открываемый на томе файл. В выполнении *ReadFile* ядро участвует в той же мере, что и в выполнении *CreateFile*, но в этом случае системному сервису *NtReadFile* не приходится искать имя — он вызывает диспетчер объектов для трансляции описателя, переданного *ReadFile*, в указатель на объект «файл». Если описатель открытого файла указывает на наличие у вызывающего потока прав на чтение файла, *NtReadFile* создает IRP типа IRP_MJ_READ и посылает его драйверу файловой системы, в которой находится файл. *NtReadFile* получает объект FSD, хранящийся в объекте «файл», и вызывает *IoCallDriver*. Диспетчер ввода-вывода находит FSD с помощью объекта FSD и передает IRP этому драйверу файловой системы.

Если считываемый файл может быть кэширован (т. е. при открытии файла в функцию *CreateFile* не передан флаг FILE_FLAG_NO_BUFFERING), FSD проверяет, инициировано ли кэширование для этого объекта «файл». Если да, поле *PrivateCacheMap* объекта «файл» указывает на структуру закрытой карты кэша (см. главу 11). В ином случае поле *PrivateCacheMap* будет пустым. Кэширование объекта «файл» инициируется FSD при первой операции записи или чтения над этим объектом, для чего FSD вызывает функцию *CcInitializeCacheMap* диспетчера кэша, и диспетчер кэша создает закрытую и общую карты кэша, а также объект «раздел» (если это еще не сделано).

Убедившись, что кэширование файла разрешено, FSD копирует данные запрошенного файла из виртуальной памяти диспетчера кэша в буфер, указатель на который передан функции *ReadFile* вызывающим потоком. Файловая система выполняет копирование в рамках блока try/except, что позволяет перехватывать все ошибки, которые могут возникнуть, если приложение указало неверный буфер. Для копирования файловая система использует функцию *CcCopyRead* диспетчера кэша, которая принимает в качестве параметров объект «файл», смещение внутри файла и длину данных.

Диспетчер кэша, выполняя *CcCopyRead*, получает указатель на общую карту кэша, хранящуюся в объекте «файл». Вспомните из главы 11, что эта карта хранит указатели на блоки управления виртуальными адресами (VACB) и что один элемент VACB соответствует 256-килобайтному блоку файла. Если

VACB-указатель для считываемой части файла пуст, *CcCopyRead* создает VACB, резервируя в виртуальном адресном пространстве диспетчера кэша 256-килобайтное представление, и проецирует на это представление указанную порцию файла (с помощью *MmMapViewInSystemCache*). Затем *CcCopyRead* просто копирует данные файла из спроецированного представления в переданный ей буфер (буфер, изначально переданный в *ReadFile*). Если файловых данных в физической памяти нет, операция копирования вызывает ошибки страниц, обслуживаемые *MmAccessFault*.

Когда возникает ошибка страницы, *MmAccessFault* изучает виртуальный адрес, вызвавший ошибку, и находит дескриптор виртуального адреса (VAD) в дереве VAD вызвавшего ошибку процесса (подробнее о дереве VAD см. главу 7). В данном случае VAD описывает представление считываемого файла, проецируемое диспетчером кэша, поэтому для обработки ошибки страницы, вызванной действительным виртуальным адресом, *MmAccessFault* вызывает *MiDispatchFault*, которая сначала находит область управления (на нее указывает VAD) и уже через нее отыскивает объект «файл», представляющий открытый файл. (Если файл открывался более чем один раз, возможно наличие списка объектов «файл», связанных указателями в закрытых картах кэша.)

Найдя объект «файл», *MiDispatchFault* вызывает функцию *IoPageRead* диспетчера ввода-вывода, чтобы создать IRP (типа IRP_MJ_READ), и посылает этот IRP к FSD, владеющему объектом «устройство», на который указывает объект «файл». Таким образом, осуществляется повторный вход в файловую систему для чтения данных, запрошенных через *CcCopyRead*, но на этот раз в IRP присутствует флаг, который сообщает о необходимости некэшируемого и связанного с подкачкой ввода-вывода. Этот флаг сигнализирует FSD, что он должен извлечь данные непосредственно с диска, и тот так и поступает, определяя, какие кластеры диска содержат запрошенные данные, и посылая соответствующие IRP диспетчеру томов, владеющему объектом тома, на котором находится файл. Поле блока параметров тома (VPB) объекта FSD указывает на объект тома.

Диспетчер виртуальной памяти ждет, когда FSD завершит чтение, а потом возвращает управление диспетчеру кэша, который продолжает операцию копирования, прерванную ошибкой страницы. По окончании работы *CcCopyRead* драйвер файловой системы возвращает управление потоку, вызвавшему *NtReadFile*; на этот момент данные из запрошенного файла уже скопированы в буфер потока.

WriteFile работает аналогичным образом с тем исключением, что системный сервис *NtWriteFile* генерирует IRP типа IRP_MJ_WRITE, а FSD вызывает не *CcCopyRead*, а *CcCopyWrite*. Последняя, как и *CcCopyRead*, проверяет, спроецированы ли на кэш части записываемого файла, и копирует в кэш содержимое буфера, переданного в *WriteFile*.

Если файловые данные уже хранятся в системном рабочем наборе, вышеописанный сценарий немного меняется. Если файловые данные находятся в кэше, *CcCopyRead* не вызывает ошибки страниц. Кроме того, в определенных обстоятельствах *NtReadFile* и *NtWriteFile* — вместо того чтобы немедлен-

но создать IRP и послать его FSD — вызывают в FSD точку входа для быстрого ввода-вывода. Вот некоторые из этих обстоятельств: считываемая часть файла должна находиться в первых 4 Гб файла, в файле не должно быть блокировок, а считываемая или записываемая часть файла не должна выходить за пределы его текущей длины.

В большинстве FSD точки быстрого ввода-вывода вызывают в диспетчере кэша функции *CcFastCopyRead* и *CcFastCopyWrite* для чтения и записи. Эти варианты стандартных процедур копирования требуют, чтобы перед копированием файловые данные были спроецированы на кэш файловой системы. Если это условие не выполнено, *CcFastCopyRead* и *CcFastCopyWrite* сообщают о невозможности быстрого ввода-вывода; тогда функции *NtReadFile* и *NtWriteFile* возвращаются к созданию IRP. (Более полное описание быстрого ввода-вывода см. в разделе «Быстрый ввод-вывод» главы 11.)

Подсистемы записи модифицированных и спроецированных страниц

Для сброса модифицированных страниц при нехватке доступной памяти периодически пробуждаются потоки принадлежащих диспетчеру памяти подсистем записи модифицированных и спроецированных страниц. Эти потоки вызывают функцию *IoSynchronousPageWrite*, чтобы создать IRP типа IRP_MJ_WRITE и записать страницы либо в страничный файл, либо в файл, модифицированный после того, как он был спроецирован. Как и в IRP, создаваемом *MiDispatchFault*, в этих IRP устанавливается флаг некэшируемого и связанного с подкачкой ввода-вывода. Поэтому для записи содержимого памяти на диск FSD обходит кэш файловой системы и выдает IRP непосредственно драйверу устройства внешней памяти.

Подсистема отложенной записи

Поток подсистемы отложенной записи, принадлежащей диспетчеру кэша, тоже участвует в записи модифицированных страниц, поскольку периодически сбрасывает на диск измененные представления разделов файлов, проецируемых на кэш. Операция сброса, выполняемая диспетчером кэша вызовом *MmFlushSection*, заставляет диспетчер памяти записать на диск все модифицированные страницы в сбрасываемой части раздела. Как и подсистемы записи модифицированных и спроецированных страниц, *MmFlushSection* посылает данные FSD через *IoSynchronousPageWrite*.

Поток, выполняющий опережающее чтение

Диспетчер кэша включает поток, отвечающий за попытку чтения данных из файлов до того, как их явным образом запросит приложение, драйвер или системный поток. Чтобы определить объем подлежащих чтению данных, поток опережающего чтения использует хронологию операций чтения, которая хранится в закрытой карте кэша объекта «файл». Выполняя опережающее чтение, этот поток просто проецирует на кэш ту часть файла, которую он хочет считать (при необходимости создавая VACB), и обращается к спроецированным данным. Если при попытках обращения возникают ошибки

страниц, активизируется обработчик ошибок страниц, который подгружает нужные страницы в системный рабочий набор.

Обработчик ошибок страниц

Мы описывали использование обработчика ошибок страниц в контексте явного файлового ввода-вывода и опережающего чтения диспетчера кэша. Но этот обработчик активизируется и всякий раз, когда приложение обращается к виртуальной памяти, являющейся представлением проецируемого файла, и встречает страницы, которые представляют часть файла, но не входят в рабочий набор приложения. Обработчик *MmAccessFault* диспетчера памяти предпринимает те же действия, что и при генерации ошибок страниц в результате выполнения *CcCopyRead* или *CcCopyWrite*, и через *IoPageRead* посылает соответствующие IRP файловой системе, в которой хранится нужный файл.

Драйверы фильтров файловой системы

Драйвер фильтра, занимающий в иерархии более высокий уровень, чем драйвер файловой системы, называется *драйвером фильтра файловой системы* (file system filter driver). (О драйверах фильтров см. главу 9.) Его способность видеть все запросы к файловой системе и при необходимости модифицировать или выполнять их, делает возможным создание таких приложений, как службы репликации удаленных файлов, шифрования файлов, резервного копирования и лицензирования. В любой коммерческий антивирусный сканер, проверяющий файлы на «лету», входит драйвер файловой системы, который перехватывает IRP-пакеты с командами IRP_MJ_CREATE, выдаваемыми при каждом открытии файла приложением. Прежде чем передать такой IRP драйверу файловой системы, которому адресована данная команда, антивирусный сканер проверяет открываемый файл на наличие вирусов. Если файл чист, антивирусный сканер передает IRP дальше по цепочке, но если файл заражен, сканер обращается к своему сервисному процессу для удаления или лечения этого файла. Если вылечить файл нельзя, драйвер фильтра отклоняет IRP (обычно с ошибкой «доступ запрещен»), чтобы вирус не смог активизироваться.

В этом разделе мы опишем работу двух специфических драйверов фильтров файловой системы: Filemon и System Restore. Filemon — утилита для мониторинга активности файловой системы (с сайта *www.sysinternals.com*), используемая во многих экспериментах в этой книге, — является примером пассивного драйвера фильтра, который не модифицирует поток IRP между приложениями и драйверами файловой системы. System Restore (Восстановление системы) — функциональность, введенная в Windows XP, — использует драйвер фильтра файловой системы для наблюдения за изменениями в ключевых системных файлах и создает их резервные копии, чтобы эти файлы можно было возвращать в те состояния, которые были у них в моменты создания точек восстановления.

ПРИМЕЧАНИЕ В Windows XP Service Pack 2 и Windows Server 2003 включен Filesystem Filter Manager (\Windows\System32\Drivers\Fltmgr.sys) как часть модели «порт-минипорт» для драйверов фильтров файловой системы. Этот компонент будет доступен и для Windows 2000. Filesystem Filter Manager кардинально упрощает разработку драйверов фильтров, предоставляя интерфейс минипорт-драйверов фильтров к подсистеме ввода-вывода Windows, а также поддерживая сервисы для запроса имен файлов, подключения к томам и взаимодействия с другими фильтрами. Компании-разработчики, в том числе Microsoft, будут писать новые фильтры файловых систем на основе инфраструктуры, предоставляемой Filesystem Filter Manager, и переносить на нее существующие фильтры.

Filemon

Утилита Filemon извлекает драйвер устройства «фильтр файловой системы» (Filem.sys) из своего исполняемого образа (Filemon.exe) при первом ее запуске после загрузки, устанавливает этот драйвер в памяти, а затем удаляет его образ с диска. Через GUI утилиты Filemon вы можете указывать ей вести мониторинг активности файловой системы на локальных томах, общих сетевых ресурсах, именованных каналах и почтовых ящиках (mail slots). Получив команду начать мониторинг тома, драйвер создает объект «устройство» фильтра и подключает его к объекту «устройству», который представляет смонтированную файловую систему на томе. Например, если драйвер NTFS смонтировал том, драйвер Filemon подключит к объекту «устройство» данного тома свой объект «устройство», используя функцию *IoAttachDeviceToDeviceStackSafe* диспетчера ввода-вывода. После этого IRP, адресованный нижележащему объекту «устройство», перенаправляется диспетчером ввода-вывода драйверу, которому принадлежит подключенный объект «устройство» (в данном случае — Filemon).

Перехватив IRP, драйвер Filemon записывает информацию о команде в этом IRP, в том числе имя целевого файла и другие параметры, специфичные для команды (например, размер и смещение считываемых или записываемых данных), в буфер режима ядра, созданный в неподкачиваемой памяти. Дважды в секунду Filemon GUI посылает IRP объекту «устройство» интерфейса Filemon, который запрашивает копию буфера, содержащего сведения о самых последних операциях, и отображает их в окне вывода. Применение Filemon подробно описывается в разделе «Анализ проблем в файловой системе» далее в этой главе.

System Restore

System Restore (Восстановление системы) — сервис, в рудиментарной форме впервые появившийся в Windows Me (Millennium Edition), позволяет восстанавливать систему до предыдущего известного состояния в ситуациях, в которых иначе пришлось бы переустанавливать какое-то приложение или даже

всю операционную систему. (System Restore нет в Windows 2000 и Windows Server 2003.) Например, если вы установили одно или несколько приложений или внесли какие-то изменения в реестр либо системный файл и это вызвало проблемы в работе приложений или крах системы, вы можете использовать System Restore для отката системных файлов и реестра в предыдущее состояние. System Restore особенно полезен, когда вы устанавливаете приложение, вносящее нежелательные изменения в системные файлы. Программы установки, совместимые с Windows XP, интегрируются с System Restore и создают точку восстановления до начала процесса установки.

Ядро System Restore содержится в сервисе SrService, который вызывается из DLL (\Windows\System32\Srsvc.dll), выполняемой в экземпляре универсального процесса — хоста сервисов (\Windows\System32\Svchost.exe). (Описание Svchost см. в главе 4.) Роль этого сервиса заключается как в автоматическом создании точек восстановления, так и в экспорте соответствующего API другим приложениям, например программам установки. (Кстати, вы можете вручную инициировать создание точки восстановления.) System Restore считывает свои конфигурационные параметры из раздела реестра HKLM\System\CurrentControlSet\Services\SR\Parameters, в том числе указывающие, сколько места на диске должно быть свободно для работы этого сервиса и через какой интервал следует автоматически создавать точки восстановления. По умолчанию данный сервис создает точку восстановления перед установкой неподписанного драйвера устройства и через каждые 24 часа работы системы. Если в упомянутом выше разделе реестра задан DWORD-параметр RPGlobalInterval, он переопределяет этот интервал и задает минимальное время в секундах между моментами автоматического создания точек восстановления.

Создавая новую точку восстановления, сервис System Restore сначала создает каталог для этой точки, затем делает снимок состояния критически важных системных файлов, включая кусты реестра, относящиеся к системе и пользовательским профилям, конфигурационную информацию WMI, файл метабазы IIS (если IIS установлена) и регистрационную базу данных COM. Потом драйвер восстановления системы, \Windows\System32\Drivers \Sr.sys, начинает отслеживать изменения в файлах и каталогах, сохраняя копии удаляемых или модифицируемых файлов в точке восстановления, а также отмечая другие изменения, например создание и удаление каталогов, в журнале регистрации изменений.

Данные точки восстановления поддерживаются для каждого тома индивидуально, поэтому журналы изменений и сохраненные файлы хранятся в каталогах \System Volume Information_restore{*XX-XXX-XXX*}, где символы *X* представляют назначенный системой GUID соответствующего тома. Такой каталог содержит подкаталоги точек восстановления с именами в виде RP*n*, где *n* — уникальный идентификатор точки восстановления. Файлы, входящие в начальный снимок точки восстановления, хранятся в каталоге Snapshot точки восстановления.

Резервным копиям файлов, созданным драйвером System Restore, присваиваются уникальные имена (вроде A0000135.dll), и эти файлы помещаются

в соответствующий каталог. С точкой восстановления может быть сопоставлено несколько журналов изменений, каждый из которых получает имя в виде Change.log.*N*, где *N* — уникальный идентификатор журнала изменений. В каждой записи этого журнала содержатся данные, которые позволяют отменить изменения, внесенные в какой-либо файл или каталог. Например, если вы удалите файл, то соответствующая этой операции запись будет хранить имя его копии в точке восстановления (допустим, A0000135.dll), а также исходные длинное и краткое имена этого файла. Сервис System Restore создает новый журнал изменений, когда размер текущего достигает 1 Мб. Рис. 12-10 отражает ход обработки запросов к файловой системе в то время, как драйвер System Restore обновляет точку восстановления, реагируя на какие-либо изменения.

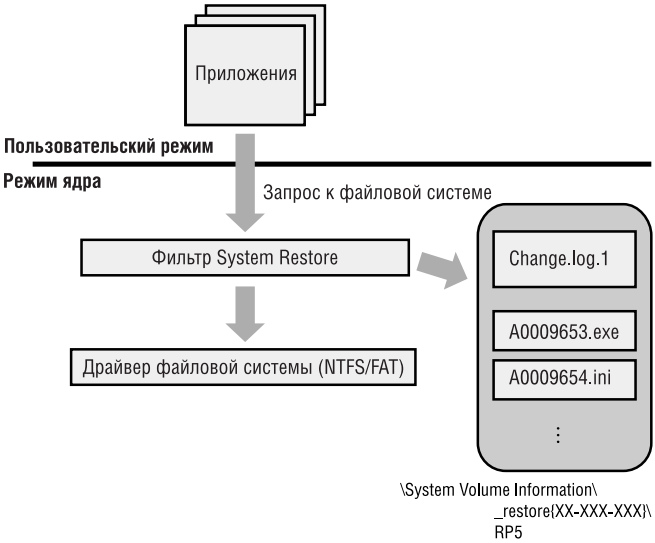


Рис. 12-10. *Так работает драйвер фильтра System Restore*

Рис. 12-11 иллюстрирует содержимое каталога, созданного System Restore и включающего несколько подкаталогов точек восстановления, а также содержимое подкаталога, который соответствует точке восстановления 1. Заметьте, что каталоги \System Volume Information недоступны ни по пользовательской учетной записи, ни по учетной записи администратора — к ним можно обращаться только по учетной записи Local System. Чтобы увидеть этот каталог, используйте утилиту PsExec с сайта *www.sysinternals.com*, как показано ниже:

```
C:\WINDOWS\SYSTEM32>psexec -s cmd

PsExec v1.55 - Execute processes remotely
Copyright (C) 2001-2004 Mark Russinovich
Sysinternals - www.sysinternals.com
```

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\WINDOWS\system32>cd \system*

C:\System Volume Information>cd _rest*

C:\System Volume Information\_restore{987E0331-0F01-427C-A58A-7A2E4AABF84D}>
```

Попав в каталог System Restore, вы можете просмотреть его содержимое командой DIR или перейти в подкаталоги, сопоставленные с точками восстановления.

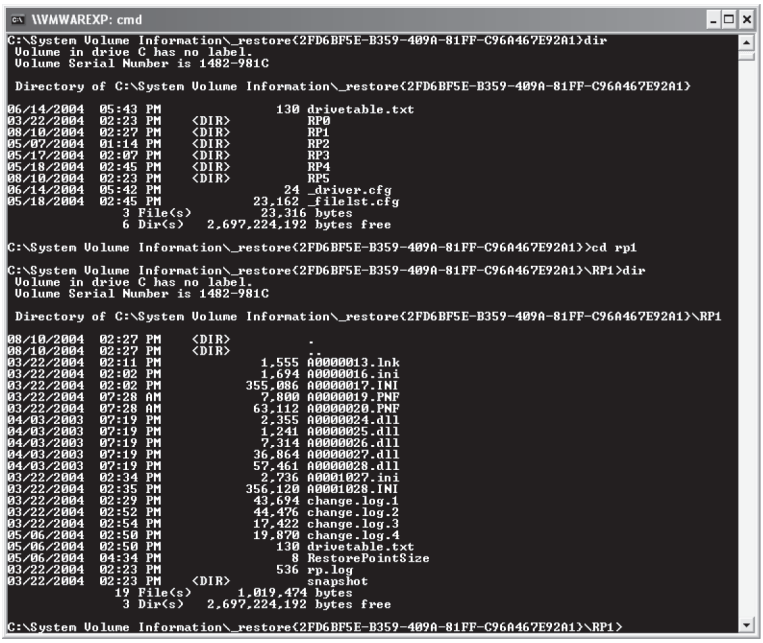


Рис. 12-11. *Каталог, созданный System Restore, и содержимое точки восстановления*

Каталог точки восстановления на загрузочном томе также содержит двоичный файл _filelst.cfg, который включает расширения файлов, изменения в которых следует фиксировать в точке восстановления, и список каталогов (хранящих, например, временные файлы), изменения в которых следует игнорировать. Этот список, документированный в Platform SDK, указывает System Restore отслеживать только файлы, отличные от файлов данных. (Вряд ли вам понравится, если при откате системы из-за проблемы с каким-нибудь приложением сервис System Restore удалит важный для вас документ Microsoft Word.)

ЭКСПЕРИМЕНТ: изучение объектов «устройство», принадлежащих фильтру System Restore

Для мониторинга изменений в файлах и каталогах драйвер фильтра System Restore должен подключить объекты «устройство» фильтра к объектам «устройство» FAT и NTFS, представляющим тома. Кроме того, он подключает объект «устройство» фильтра к объектам «устройство», представляющим драйверы файловой системы, чтобы узнавать о монтировании новых томов файловой системой и соответственно подключать к ним объекты «устройство» фильтра. Объекты «устройство» System Restore можно просмотреть с помощью отладчика ядра:

```
lkd> !drvobj \filesystem\sr
Driver object (81543850) is for:
 \FileSystem\sr
Driver Extension List: (id , addr)

Device Object list:
814ee370  81542dd0  81543728
```

В этом примере вывода у драйвера System Restore три объекта «устройство». Последний из них в списке называется *SystemRestore* — он служит интерфейсом, которому компоненты System Restore пользовательского режима направляют свои команды:

```
lkd> !devobj 81543728
Device object (81543728) is for:
 SystemRestore \FileSystem\sr DriverObject 81543850
Current Irp 00000000 RefCount 1 Type 00000022 Flags 00000040
Dacl e128feac DevExt 00000000 DevObjExt 815437e0
ExtensionFlags (0x80000000)  DOE_DESIGNATED_FDO
Device queue is not busy.
```

Первый и второй объекты подключены к объектам «устройство» файловой системы NTFS:

```
lkd> !devobj 814ee370
Device object (814ee370) is for:
  \FileSystem\sr DriverObject 81543850
Current Irp 00000000 RefCount 0 Type 00000008 Flags 00000000
DevExt 814ee428 DevObjExt 814ee570
ExtensionFlags (0x80000000)  DOE_DESIGNATED_FDO
AttachedTo (Lower) 81532020 \FileSystem\Ntfs
Device queue is not busy.
lkd> !devobj 81542dd0
Device object (81542dd0) is for:
  \FileSystem\sr DriverObject 81543850
Current Irp 00000000 RefCount 0 Type 00000008 Flags 00000000
DevExt 81542e88 DevObjExt 81542fd0
ExtensionFlags (0x80000000)  DOE_DESIGNATED_FDO
```

см. след. стр.

```
AttachedTo (Lower) 815432e8 \FileSystem\Ntfs
Device queue is not busy.
```

Один из объектов «устройство» NTFS является интерфейсом для драйвера файловой системы NTFS, так как его имя — *NTFS*:

```
lkd> !devobj 815432e8
Device object (815432e8) is for:
 Ntfs \FileSystem\Ntfs DriverObject 81543410
Current Irp 00000000 RefCount 1 Type 00000008 Flags 00000040
Dacl e1297154 DevExt 00000000 DevObjExt 815433a0
ExtensionFlags (0x80000000)  DOE_DESIGNATED_FDO
AttachedDevice (Upper) 81542dd0 \FileSystem\sr
Device queue is not busy.
```

Другой из них представляет смонтированный NTFS-том на C:, и, поскольку в системе только один том, у этого объекта нет имени:

```
lkd> !devobj 81532020
Device object (81532020) is for:
  \FileSystem\Ntfs DriverObject 81543410
Current Irp 00000000 RefCount 0 Type 00000008 Flags 00000000
DevExt 815320d8 DevObjExt 81532880
ExtensionFlags (0x80000000)  DOE_DESIGNATED_FDO
AttachedDevice (Upper) 814ee370 \FileSystem\sr
Device queue is not busy.
```

Когда пользователь сообщает системе о необходимости восстановления, мастер System Restore (\Windows\System32\Restore\Rstrui.exe) создает параметр RestoreInProgress типа DWORD в разделе реестра System Restore и присваивает ему значение 1. Затем он инициирует перезагрузку системы, вызывая Windows-функцию *ExitWindowsEx*. После перезагрузки процесс Winlogon (\Windows\System32\Winlogon.exe) обнаруживает, что нужно выполнить восстановление, копирует сохраненные файлы в исходные каталоги и с помощью файлов журнала отменяет изменения, внесенные в системные файлы и каталоги. По окончании этого процесса загрузка системы возобновляется. Перезагрузка необходима не только для большей безопасности восстановления, но и для активизации восстановленных кустов реестра.

В Platform SDK документированы две API-функции System Restore, *SRSetRestorePoint* и *SRRemoveRestorePoint*, предназначенные для программ установки. Разработчики должны продумать расширения файлов, используемые их приложениями, чтобы они не сохраняли пользовательские данные в файлах тех типов, которые защищаются сервисом System Restore. Иначе пользователь может потерять свои данные при откате системы до точки восстановления.

Анализ проблем в файловой системе

В главе 4 было показано, как система и приложения хранят данные в реестре. Если в реестре появляются какие-то проблемы, скажем, из-за неправильно

сконфигурированной защиты или недостающих параметров/разделов реестра, то они могут быть причиной многих сбоев в работе системы и приложений. Помимо этого, система и приложения используют файлы для хранения данных и обращаются к образам DLL и исполняемых файлов. Значит, ошибки в конфигурации защиты NTFS и отсутствие каких-либо файлов или каталогов также являются распространенной причиной сбоев в работе системы и приложений. А все потому, что система и приложения часто полагаются на возможность беспрепятственного доступа к таким файлам и начинают вести себя непредсказуемым образом, если эти файлы оказываются недоступны.

Утилита Filemon отражает все файловые операции по мере их выполнения, что превращает ее в идеальный инструмент для анализа сбоев системы и приложений из-за проблем в файловой системе. Пользовательский интерфейс Filemon практически идентичен таковому в Regmon, и Filemon включает те же средства фильтрации, выделения и поиска, что и Regmon. Первый запуск Filemon в системе требует учетной записи с теми же привилегиями, что и в случае Regmon: Load Driver и Debug. После загрузки драйвер остается резидентным в памяти, поэтому для последующих запусков Filemon достаточно привилегии Debug.

Базовый и расширенный режимы Filemon

При запуске Filemon начинает работу в базовом режиме, в котором отображаются те операции в файловой системе, которые наиболее полезны для анализа проблем. В этом режиме Filemon не показывает определенные операции в файловой системе, в том числе:

- обращения к файлам метаданных NTFS;
- операции в процессе System;
- ввод-вывод, связанный со страничным файлом;
- ввод-вывод, генерируемый процессом Filemon;
- неудачные попытки быстрого ввода-вывода.

Кроме того, в базовом режиме Filemon сообщает об операциях файлового ввода-вывода, используя описательные имена, а не типы IRP, которые на самом деле представляют их. В частности, операции IRP_MJ_WRITE и FASTIO_WRITE отображаются как Write, а операции IRP_MJ_CREATE — как Open (при открытии существующих файлов) или Create (при создании новых файлов).

ЭКСПЕРИМЕНТ: наблюдение за активностью файловой системы в простаивающей системе

Драйверы файловых систем Windows поддерживают *уведомление об изменениях в файлах*, что позволяет приложениям узнавать о таких изменениях в файловой системе без постоянного ее опроса. Для этого предназначены Windows-функции *ReadDirectoryChangesW*, *FindFirstChangeNotification* и *FindNextChangeNotification*. Таким образом, запуская Filemon в простаивающей системе, вы не увидите повторяющихся обращений к файлам и каталогам, поскольку такая активность не

см. след. стр.

обязательно должна негативно отразиться на общей производительности системы.

Запустите Filemon и через несколько секунд проверьте в журнале вывода, есть ли попытки периодического опроса. Обнаружив соответствующую строку, щелкните ее правой кнопкой мыши и выберите из контекстного меню Process Properties для просмотра детальных сведений о процессе, который выполняет операции опроса.

Методики анализа проблем с применением Filemon

Два основных метода анализа проблем с применением Filemon идентичны таковым при использовании Regmon: поиск последней операции в трассировочной информации Filemon перед тем, как в приложении произошел сбой, или сравнение трассировочной информации Filemon для сбойного приложения с аналогичными сведениями для работающей системы. Подробнее об этих способах см. раздел «Методики анализа проблем с применением Regmon» главы 4.

Обращайте внимание на записи в выводе Filemon со значениями FILE NOT FOUND, NO SUCH FILE, PATH NOT FOUND, SHARING VIOLATION и ACCESS DENIED в столбце Result. Первые три значения указывают на то, что приложение или система пытается открыть несуществующий файл или каталог. Во многих случаях эти ошибки не свидетельствуют о серьезной проблеме. Например, если вы запускаете какую-то программу из диалогового окна Run (Запуск программы), не задавая полный путь к ней, Explorer будет искать эту программу в каталогах, перечисленных в переменной окружения PATH, пока не найдет нужный образ или не закончит просмотр всех перечисленных каталогов. Каждая попытка найти образ в каталоге, где такого образа нет, отражается в выводе Filemon строкой, аналогичной приведенной ниже:

```
5:28:26 PMEXPLORER.EXE:1568FASTIO_QUERY_OPENC:\Documents and Settings\
mark.AUSTIN\Start Menu\test.exe    FILE NOT FOUND    Attributes: Error
```

Ошибки, связанные с отклонением попыток доступа, — частая причина сбоев приложений при работе с файловой системой, и они возникают, когда у приложения нет соответствующего разрешения на открытие файла или каталога. Некоторые приложения не проверяют коды ошибок или не обрабатывают ошибки, из-за чего происходит их крах или аварийное завершение, а некоторые выводят при этом сообщения о других ошибках, которые лишь маскируют истинную причину неудачи файловой операции.

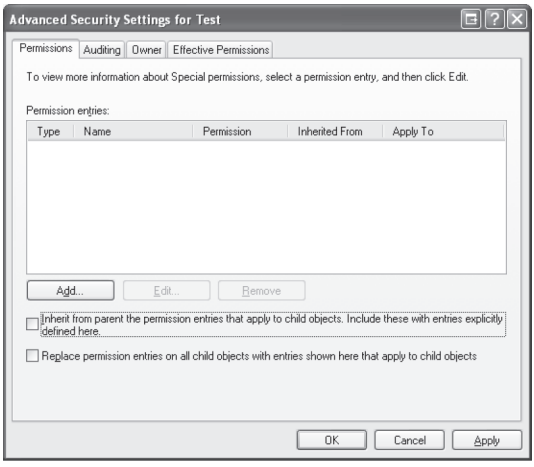
Прорехи в защите из-за переполнения буфера представляют серьезную угрозу безопасности, но код результата BUFFER OVERFLOW — просто способ, используемый драйвером файловой системы, чтобы сообщить приложению о нехватке места в выделенном буфере для сохранения полученных данных. Разработчики приложений применяют этот способ, чтобы определить правильный размер буфера, так как драйвер файловой системы заодно сообщает и эту информацию. За операциями с кодом результата BUFFER OVERFLOW обычно следуют операции с успешным результатом.

ЭКСПЕРИМЕНТ: выявление истинной причины ошибки с помощью Filemon

Иногда приложения выводят сообщения об ошибках, которые не раскрывают истинную причину проблемы. Такие сообщения могут запутать и заставить вас тратить время на диагностику несуществующих проблем. Если сообщение об ошибке относится к проблеме в файловой системе, Filemon покажет, какие нижележащие ошибки могли бы предшествовать этому сообщению.

В данном эксперименте вы установите разрешение для каталога, а затем попытаетесь выполнить в нем операцию сохранения файла из Notepad, что приведет к появлению сообщения о совсем другой ошибке. Filemon покажет истинную ошибку и причину вывода в Notepad именно такого сообщения.

1. Запустите Filemon и установите включающий фильтр для «notepad.exe».
2. Откройте Explorer и создайте каталог Test в одном из каталогов NTFS-тома. (В данном примере используется корневой каталог.)
3. Измените разрешения для каталога Test так, чтобы запретить любой вид доступа к нему. Не исключено, что вам придется открыть диалоговое окно Advanced Security Settings (Дополнительные параметры безопасности) и задать параметры на вкладке Permissions (Разрешения), чтобы удалить наследуемые разрешения защиты.



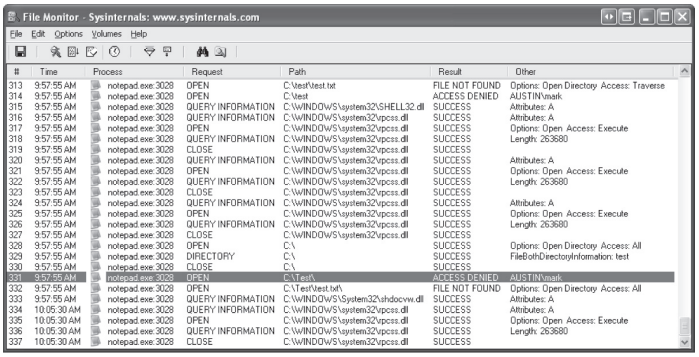
Когда вы примените измененные разрешения, Explorer должен будет предупредить вас о том, что никто не получит доступа к этой папке.

4. Запустите Notepad (Блокнот) и введите в нем какой-нибудь текст. Потом выберите команду Save (Сохранить) из меню File (Файл). В поле File Name (Имя файла) диалогового окна Save As (Сохранить как) введите **c:\test\test.txt** (если вы создали папку на томе C:).

см. след. стр.

5. Notepad выведет следующее сообщение об ошибке.

6. Это сообщение подразумевает, что каталог C:\Test не существует.
7. В трассировочной информации Filemon вы должны увидеть примерно то же, что и на следующей иллюстрации.



Вывод в нижней части иллюстрации был сгенерирован непосредственно перед появлением этого сообщения об ошибке. Строка 331 показывает, что Notepad пытался открыть C:\Test и получил отказ в доступе. Сразу после этого, как видно из строки 332, он попытался открыть каталог C:\Test\Test.txt (на это указывает флаг Directory в столбце Other на той же строке) и получил ошибку «файл не найден», потому что такого каталога нет. Сообщение Notepad «Path does not exist» (Путь не существует) согласуется с последней ошибкой, но не с первой. Поэтому кажется, что Notepad сначала пытался открыть каталог, а когда это не удалось, он почему-то решил, что имя C:\Test\Test.txt соответствует каталогу, а не файлу. Не сумев открыть такой каталог, Notepad вывел сообщение об ошибке, но истинной причиной, как показывает Filemon, был отказ в доступе.

Filemon широко применяется Microsoft и другими организациями для решения трудных или почти неуловимых проблем. Рассмотрим один из примеров, где Filemon помог докопаться до истинной причины некорректного сообщения об ошибке, генерируемого службой Windows Installer. Когда пользователь пытался установить программу из ее файла Windows Installer Package, служба Windows Installer сообщала об ошибке, показанной на рис. 12-12; в нем утверждалось, что этой службе не удается записать что-либо в папку Temp. Пользователь убедился, что вопреки этому утверждению каталог, выделенный для временных файлов и заданный в его профиле (эту информацию он по-

лучил, набрав в консольном окне команду set temp), находится на томе, где достаточно свободного места, и имеет разрешения по умолчанию.

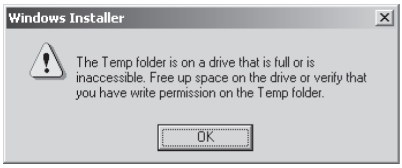


Рис. 12-12. *Сообщение об ошибке от Microsoft Windows Installer*

Тогда пользователь запустил Filemon для трассировки операций в файловой системе, которые приводят к этой ошибке, и обнаружил ее причину (см. выделенную строку на рис. 12-13). Из этих данных стало ясно, что служба Windows Installer обращалась не к каталогу временных файлов, заданному в профиле пользователя, а к \Windows\Installer. Строка со значением ACCESS DENIED указала на то, что служба Windows Installer выполнялась под учетной записью локальной системы, поэтому пользователь так изменил разрешения на доступ к \Windows\Installer, чтобы учетная запись локальной системы предоставляла права на доступ к этому каталогу для записи. Это и решило проблему.

Рис. 12-13. *Вывод Filemon для службы Microsoft Windows Installer*

Еще один пример анализа проблем с помощью Filemon. В этом случае пользователь запускал Microsoft Word, и буквально через несколько секунд набора текста окно Word закрывалось без всякого уведомления. Трассировочная информация Filemon, часть которой представлена на рис. 12-14, показала, что перед самым завершением Word повторно считывал одну и ту же часть файла с именем Mssp3es.lex. (Когда процесс завершается, система автоматически закрывает все открытые им описатели. Именно это и отражают строки с номерами от 25460.) Пользователь выяснил, что файлы с расширением .lex относятся к Microsoft Office Proofing Tools, и переустановил этот компонент, после чего проблема исчезла.

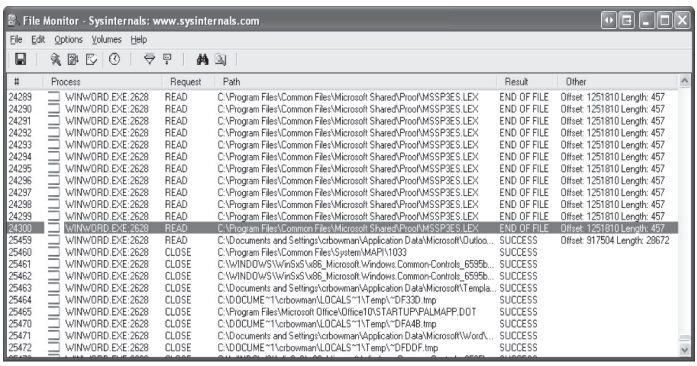


Рис. 12-14. *Microsoft Word читает файл с расширением .lex*

В третьем примере при каждом запуске Microsoft Excel выводилось сообщение об ошибке, показанное на рис. 12-15. Из трассировочной информации Filemon на рис. 12-16, полученной в ходе запуска Excel, обнаружилось, что Excel считывает файл 59403e20 из подкаталога Xlstart каталога, в который установлен Microsoft Office. Пользователь изучил документацию на Excel и выяснил, что Excel пытается автоматически открывать любые файлы, хранящиеся в каталоге Xlstart. Однако этот файл не имел никакого отношения к Excel, поэтому открыть его не удавалось и в итоге появлялось сообщение об ошибке. Удаление этого файла устранило проблему.

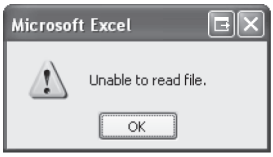


Рис. 12-15. *Сообщение об ошибке при запуске Microsoft Excel*

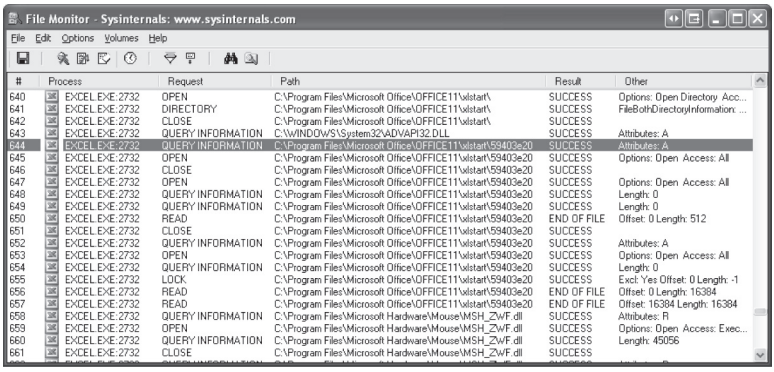


Рис. 12-16. *Трассировка в Filemon процесса запуска Microsoft Excel*

Последний пример связан с выявлением устаревших DLL. Пользователь запускал Microsoft Access 2000, и эта программа зависала, как только он пытался импортировать какой-нибудь файл Microsoft Excel. В другой системе с Microsoft Access 2000 тот же файл импортировался успешно. После трассировки операции импорта в обеих системах файлы журналов сравнивались с помощью Windiff. Результаты сравнения представлены на рис. 12-17.

Отбросив незначимые расхождения вроде разных имен временных файлов (строка 19) и разный регистр букв в именах файлов (строка 26), пользователь обнаружил первое существенное различие в результатах трассировки в строке 37. В системе, где импорт заканчивался неудачей, Microsoft Access загружал копию Accwiz.dll из каталога \Winnt\System32, тогда как в системе, где импорт проходил успешно, эта программа считывала Accwiz.dll из \Progra~1\Files\Microsoft\Office. Пользователь изучил копию Accwiz.dll в \Winnt\System32 и заметил, что она относится к более старой версии Microsoft Access, но порядок поиска DLL в системе приводил к тому, что этот экземпляр обнаруживался первым и до экземпляра нужной версии в каталоге, где ус-

тановлен Microsoft Access 2000, дело не доходило. Удалив эту копию и зарегистрировав корректную версию, пользователь решил проблему.

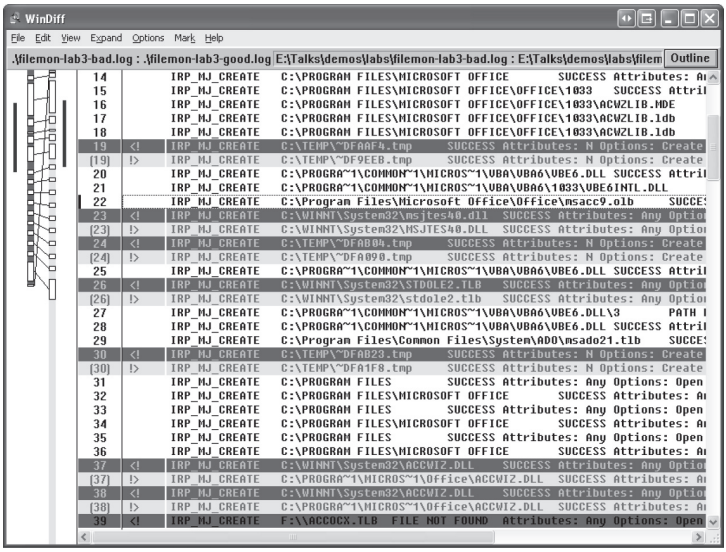


Рис. 12-17. *Сравнение журналов трассировки Microsoft Access в двух системах*

Это лишь некоторые примеры, демонстрирующие, как с помощью Filemon выявлять истинные причины проблем в файловой системе, о которых приложения не всегда сообщают корректно. В остальной части главы мы сосредоточимся на описании «родной» для Windows файловой системы — NTFS.

Цели разработки и особенности NTFS

В следующем разделе мы расскажем о требованиях, определявших разработку NTFS, и о дополнительных возможностях этой файловой системы.

Требования к файловой системе класса «high end»

С самого начала разработка NTFS велась с учетом требований, предъявляемых к файловой системе корпоративного класса. Чтобы свести к минимуму потери данных в случае неожиданного выхода системы из строя или ее краха, файловая система должна гарантировать целостность своих метаданных. Для защиты конфиденциальных данных от несанкционированного доступа файловая система должна быть построена на интегрированной модели защиты. Наконец, она должна поддерживать защиту пользовательских данных за счет программной избыточности данных в качестве недорогой альтернативы аппаратным решениям. Здесь вы узнаете, как эти возможности реализованы в NTFS.

Восстанавливаемость

В соответствии с требованиями к надежности хранения данных и доступа к ним NTFS обеспечивает восстановление файловой системы на основе кон-

цепции *атомарной транзакции* (atomic transaction). Атомарные транзакции — это метод обработки изменений в базе данных, при котором сбои в работе системы не нарушают корректности или целостности базы данных. Суть атомарных транзакций заключается в том, что некоторые операции над базой данных, называемые *транзакциями*, выполняются по принципу «все или ничего». (Транзакцию можно определить как операцию ввода-вывода, изменяющую данные файловой системы или структуру каталогов тома.) Отдельные изменения на диске, составляющие транзакцию, выполняются атомарно: в ходе транзакции на диск должны быть внесены все требуемые изменения. Если транзакция прервана аварией системы, часть изменений, уже внесенных к этому моменту, нужно отменить. Такая отмена называется *откатом* (roll back). После отката база данных возвращается в исходное согласованное состояние, в котором она была до начала транзакции.

NTFS использует атомарные транзакции для реализации возможности восстановления файловой системы. Если некая программа инициирует операцию ввода-вывода, которая изменяет структуру NTFS-тома, т. е. модифицирует структуру каталогов, увеличивает длину файла, выделяет место под новый файл и др., то NTFS обрабатывает такую операцию как атомарную транзакцию. NTFS гарантирует, что транзакция будет либо полностью выполнена, либо отменена, если хотя бы одну из операций не удастся завершить из-за сбоя системы. О том, как это делается в NTFS, см. раздел «Поддержка восстановления в NTFS» далее в этой главе.

Кроме того, NTFS использует избыточность для хранения критически важной информации файловой системы, так что, если на диске появится сбойный сектор, она все равно сможет получить доступ к этой информации. Это одна из особенностей NTFS, отличающих ее от FAT и HPFS («родной» файловой системы OS/2).

Защита

Защита в NTFS построена на модели объектов Windows. Файлы и каталоги защищены от доступа пользователей, не имеющих соответствующих прав (подробнее о защите в Windows см. главу 8). Открытый файл реализуется в виде объекта «файл» с дескриптором защиты, хранящимся на диске как часть файла. Прежде чем процесс сможет открыть описатель какого-либо объекта, в том числе объекта «файл», система защиты Windows должна убедиться, что у этого процесса есть соответствующие полномочия. Дескриптор защиты в сочетании с требованием регистрации пользователя при входе в систему гарантирует, что ни один процесс не получит доступа к файлу без разрешения системного администратора или владельца файла.

Избыточность данных и отказоустойчивость

Восстанавливаемость NTFS действительно гарантирует, что файловая система тома останется доступной, но не дает гарантии полного восстановления пользовательских файлов. Последнее возможно за счет поддержки избыточности данных.

Избыточность данных для пользовательских файлов реализуется через многоуровневую модель драйверов Windows (см. главу 9), которая поддерживает отказоустойчивые диски. При записи данных на диск NTFS взаимодействует с диспетчером томов, а тот — с драйвером жесткого диска. Диспетчер томов может *зеркалировать*, или дублировать, данные одного диска на другом и таким образом позволяет при необходимости использовать данные с избыточной копии. Поддержка таких функций обычно называется RAID уровня 1. Диспетчеры томов также могут записывать данные в чередующиеся области (stripes) на три и более дисков, используя один диск для хранения информации о четности. Если данные на одном диске потеряны или стали недоступными, драйвер может реконструировать содержимое диска с помощью логической операции XOR. Такая поддержка называется RAID уровня 5 (подробнее о чередующихся и зеркальных томах, а также о томах RAID-5 см. главу 10).

Дополнительные возможности NTFS

NTFS — это не только восстанавливаемая, защищенная, надежная и эффективная файловая система, способная работать в системах повышенной ответственности. Она поддерживает ряд дополнительных возможностей (некоторые из них доступны приложениям через API-функции, другие являются внутренними):

- множественные потоки данных;
- имена на основе Unicode;
- универсальный механизм индексации;
- динамическое переназначение плохих кластеров;
- жесткие связи и точки соединения;
- сжатие и разреженные файлы;
- протоколирование изменений;
- квоты томов, индивидуальные для каждого пользователя;
- отслеживание ссылок;
- шифрование;
- поддержка POSIX;
- дефрагментация
- поддержка доступа только для чтения.

Обзор этих возможностей приводится в следующих разделах.

Множественные потоки данных

В NTFS каждая единица информации, сопоставленная с файлом (имя и владелец файла, метка времени и т. д.), реализована в виде атрибута файла (атрибута объекта NTFS). Каждый атрибут состоит из одного потока данных (stream), т. е. из простой последовательности байтов. Это позволяет легко добавлять к файлу новые атрибуты (и соответственно новые потоки). По-

скольку данные являются всего лишь одним из атрибутов файла и поскольку можно добавлять новые атрибуты, файлы и каталоги NTFS могут содержать несколько потоков данных.

В любом файле NTFS по умолчанию имеется один безымянный поток данных. Приложения могут создавать дополнительные, именованные потоки данных и обращаться к ним по именам. Чтобы не изменять Windows-функции API ввода-вывода, которым имя файла передается как строковый аргумент, имена потоков данных задаются через двоеточие (:) после имени файла, например:

```
myfile.dat:stream2
```

Каждый поток имеет свой выделенный размер (объем зарезервированного для него дискового пространства), реальный размер (использованное число байтов) и длину действительных данных (инициализированная часть потока). Кроме того, каждому потоку предоставляется отдельная файловая блокировка, позволяющая блокировать диапазоны байтов и поддерживать параллельный доступ.

Множественные потоки данных использует, например, компонент поддержки файлового сервера Apple Macintosh, поставляемый с Windows Server. Системы Macintosh используют два потока данных в каждом файле: один — для хранения данных, другой — для хранения информации о ресурсах (тип файла, значок, представляющий файл в пользовательском интерфейсе, и т. д.). Поскольку NTFS поддерживает множественные потоки данных, пользователь Macintosh может скопировать целую папку Macintosh на сервер Windows, а другой пользователь Macintosh может скопировать ее с сервера без потери информации о ресурсах.

Windows Explorer — еще одно приложение, использующее такие потоки. Когда вы щелкаете правой кнопкой мыши файл NTFS и выбираете команду Properties, на экране появляется диалоговое окно, вкладка Summary (Сводка) которого позволяет сопоставить с файлом такую информацию, как заголовок, тема, имя автора и ключевые слова. Windows Explorer хранит эту информацию в альтернативном потоке под названием Summary Information, добавляемом к файлу.

Другие приложения тоже могут использовать множественные потоки данных. Например, утилита резервного копирования могла бы с помощью дополнительного потока данных сохранять метки времени, связанные с резервным копированием файлов, а утилита архивирования — реализовать иерархическое хранилище, в котором файлы, созданные ранее заданной даты или не востребованные в течение указанного периода, перемещались бы на ленточные накопители. При этом она могла бы скопировать файл на ленту, установить его поток данных по умолчанию равным 0 и добавить новый поток данных, указывающий название и местонахождение картриджа, в котором хранится файл.

ЭКСПЕРИМЕНТ: просмотр потоков

Большинство приложений Windows не рассчитано на работу с дополнительными именованными потоками, но команды *echo* и *more* поддерживают эту функциональность. Таким образом, самый простой способ понаблюдать за потоками данных в действии — создать именованный поток с помощью *echo*, а затем вывести его на экран с помощью *more*. В результате выполнения команд, показанных ниже, создается файл test с потоком stream.

```
C:\>echo hello > test:stream
C:\>more < test:stream
hello
C:\>
```

При перечислении содержимого каталога размер файла test не отражает данные, хранящиеся в дополнительном потоке, поскольку в этом случае NTFS возвращает размер только безымянного потока данных.

```
C:\>dir test
 Volume in drive C is WINDOWS
 Volume Serial Number is 3991-3040

 Directory of C:\

08/01/00  02:37p                     0 test
               1 File(s)              0 bytes
                            112,558,080 bytes free
```

Утилита Streams (*www.sysinternals.com*) позволяет определить, в каких файлах и каталогах есть дополнительные потоки данных:

```
C:\>streams test

Streams v1.5 - Enumerate alternate NTFS data streams
Copyright (C) 1999-2003 Mark Russinovich
Sysinternals - www.sysinternals.com
 C:\Temp\test:
          :stream:$DATA 8
```

Имена на основе Unicode

Как и Windows в целом, NTFS полностью поддерживает Unicode, используя Unicode-символы для хранения имен файлов, каталогов и томов. Unicode, 16-битная кодировка символов, обеспечивает уникальное представление любого символа основных языков мира, что упрощает обмен информацией между странами. Поскольку в Unicode имеется уникальное представление каждого символа, последний не зависит от того, какая кодовая страница загружена в

операционную систему. Длина имени каждого каталога или файла в пути может достигать 255 символов; в нем могут быть символы Unicode, пробелы и несколько точек.

Универсальный механизм индексации

Архитектура NTFS позволяет индексировать атрибуты файлов на дисковом томе. Это дает возможность файловой системе вести эффективный поиск файлов по неким критериям, например находить все файлы в определенном каталоге. Файловая система FAT индексирует имена файлов, но не сортирует их, что замедляет просмотр больших каталогов.

Некоторые функции NTFS используют преимущества универсальной индексации, в том числе консолидированных дескрипторов защиты, где дескрипторы защиты файлов и каталогов на томе хранятся в едином внутреннем потоке без дубликатов; при этом они индексируются с использованием внутреннего идентификатора защиты, определяемого NTFS. Об индексации с помощью этих средств см. раздел «Структура NTFS на диске» далее в этой главе.

Динамическое переназначение плохих кластеров

Обычно, если программа пытается считать данные из плохого сектора диска, операция чтения заканчивается неудачей, а данные соответствующего кластера становятся недоступными. Однако, если диск отформатирован как отказоустойчивый том NTFS, специальный драйвер Windows динамически считывает «хорошую» копию данных, хранившихся в плохом секторе, и посылает NTFS предупреждение о плохом секторе. NTFS выделяет новый кластер, заменяющий тот, в котором находится плохой сектор, и копирует данные в этот кластер. Плохой кластер помечается как аварийный и больше не используется. Восстановление данных и динамическое переназначение плохих кластеров особенно полезно для файловых серверов и отказоустойчивых систем, а также для всех приложений, в которых потеря данных недопустима. Если на момент появления плохого сектора диспетчер томов не был загружен, NTFS все равно заменяет кластер и не допускает его повторного использования, хотя восстановить данные из плохого сектора уже не удастся.

Жесткие связи и точки соединения

Жесткие связи (hard links) позволяют ссылаться на один и тот же файл по нескольким путям (для каталогов жесткие связи не поддерживаются). Если вы создаете жесткую связь с именем C:\Users\Documents\Spec.doc, которая ссылается на существующий файл C:\My Documents\Spec.doc, то с одним дисковым файлом будут связаны два пути, и вы сможете обращаться к этому файлу, используя оба пути. Процессы могут создавать жесткие связи вызовом Windows-функции *CreateHardLink* или POSIX-функции *ln*.

В дополнение к жестким связям NTFS поддерживает другой тип перенаправления — *точки соединения* (junctions), также называемые символьными ссылками (symbolic links). Они позволяют перенаправлять трансляцию имени файла или каталога из одного каталога в другой. Например, если путь

C:\Drivers — ссылка, которая перенаправляет в C:\Windows\System32\Drivers, то приложение, читающее C:\Drivers\Ntfs.sys, на самом деле читает C:\Windows\System\Drivers\Ntfs.sys. Точки соединения представляют собой удобное средство «подъема» каталогов, расположенных слишком глубоко в дереве каталогов, на более высокий уровень, не нарушая исходной структуры или содержимого дерева каталогов. Так, в предыдущем примере каталог драйверов «поднят» на два уровня по сравнению с реальным. Точки соединения неприменимы к удаленным каталогам — они используются только для каталогов на локальных томах.

Точки соединения опираются на механизм NTFS «точки повторного разбора» (см. раздел «Точки повторного разбора» далее в этой главе). *Точка повторного разбора* (reparse point) — это файл или каталог, с которым сопоставлен блок данных, называемых *данными повторного разбора* (reparse data); они представляют собой пользовательские данные о файле или каталоге, например о его состоянии или местонахождении. Эти данные могут быть считаны из точки повторного разбора приложением, которое создало их, драйвером файловой системы или диспетчером ввода-вывода. Обнаружив точку повторного разбора при поиске файла или каталога, NTFS возвращает *код статуса повторного разбора*, который сигнализирует драйверам фильтров файловой системы, подключенным к дисковому тому, и диспетчеру ввода-вывода о необходимости анализа данных повторного разбора. Каждый тип точек повторного разбора имеет уникальный *тэг повторного разбора* (reparse tag) — он позволяет компоненту, отвечающему за интерпретацию данных конкретной точки повторного разбора, распознавать свои точки разбора, не проверяя их данные. Далее владелец тэга повторного разбора (драйвер фильтра файловой системы или диспетчер ввода-вывода) может выбрать один из следующих вариантов дальнейших действий.

- Владелец тэга повторного разбора может манипулировать полным именем файла, при анализе которого обнаружена точка повторного разбора, и инициировать ввод-вывод по измененному пути. Точки соединения используют этот вариант, например, для перенаправления каталогов.
- Владелец тэга повторного разбора может удалить из файла точку повторного разбора, каким-либо образом изменить файл, а затем инициировать новую операцию файлового ввода-вывода. По такому принципу точки повторного разбора используются системой Hierarchical Storage Management (HSM). HSM архивирует файлы, перемещая их содержимое на ленточные накопители и оставляя вместо содержимого файлов точки повторного разбора. Когда какой-либо процесс обращается к архивированному файлу, драйвер фильтра HSM (\Windows\System32\Drivers\Rsfilter.sys) удаляет из этого файла точку повторного разбора, считывает его данные с архивного носителя и инициирует повторное обращение к файлу.

Windows-функций для создания точек повторного разбора нет. Вместо них процессы должны использовать управляющий код FSCTL_SET_REPARSE_POINT файловой системы в сочетании с Windows-функцией *DeviceIoControl*.

Процесс может запросить содержимое точки повторного разбора с помощью управляющего кода FSCTL_GET_REPARSE_POINT. В атрибутах файла, сопоставленного с точкой повторного разбора, присутствует флаг FILE_ATTRIBUTE_REPARSE_POINT, что позволяет приложениям проверять наличие точек повторного разбора вызовом Windows-функции *GetFileAttributes*.

ЭКСПЕРИМЕНТ: создание точки соединения

В Windows нет средств для создания точек соединения, но вы можете создать такую точку с помощью утилиты Junction (*wwwsysinternals.com*) или Linkd из ресурсов Windows. Утилита Linkd позволяет просмотреть определения существующих точек соединения, а Junction — вывести информацию о точках соединения и точках повторного разбора.

Сжатие и разреженные файлы

NTFS поддерживает сжатие файловых данных. Поскольку NTFS выполняет процедуры сжатия и декомпрессии прозрачно, нет необходимости модифицировать приложения для того, чтобы они могли пользоваться преимуществами этой функции. Каталог также может быть сжат, что влечет за собой сжатие и тех файлов, которые будут впоследствии созданы в этом каталоге.

Приложения сжимают и разархивируют файлы, передавая *DeviceIoControl* управляющий код FSCTL_SET_COMPRESSION. Для запроса состояния сжатия файла или каталога используется управляющий код FSCTL_GET_COMPRESSION. У сжатого файла или каталога установлен флаг FILE_ATTRIBUTE_COMPRESSED, поэтому приложения могут определять состояние сжатия файла или каталога вызовом *GetFileAttributes*.

Второй тип сжатия известен под названием *разреженные файлы* (sparse files). Если файл помечен как разреженный, NTFS не выделяет на томе место для тех частей файла, которые определены приложением как пустые. При чтении приложением пустых областей разреженного файла NTFS просто возвращает буферы, заполненные нулевыми значениями. Этот тип сжатия полезен для клиент-серверных приложений, в которых реализовано протоколирование с циклическими буферами (circular-buffer logging): сервер регистрирует информацию в файле, а клиент асинхронно считывает ее. Поскольку информация, уже считанная клиентом, больше не нужна, продолжать хранить ее в файле не требуется. Если такой файл является разреженным, клиент может определять считанные им области как пустые, тем самым освобождая место на томе. А сервер может добавлять новую информацию в файл, не опасаясь, что он в конечном счете займет все свободное пространство на томе.

Как и в случае сжатых файлов, NTFS прозрачно управляет разреженными файлами. Приложения указывают состояние разреженности файла, передавая *DeviceIoControl* управляющий код FSCTL_SET_SPARSE. Чтобы определить диапазон файла как пустой, приложения используют код FSCTL_SET_ZERO_DATA, а чтобы запросить у NTFS описание того, какие части файла являются разреженными, — код FSCTL_QUERY_ALLOCATED_RANGES. Разре-

женные файлы применяются, в частности, в журнале изменений NTFS, о котором мы расскажем в следующем разделе.

Протоколирование изменений

Приложениям многих типов нужно отслеживать изменения файлов и каталогов тома. Например, программа автоматического резервного копирования первоначально выполняет полное резервное копирование, а в дальнейшем копирует только измененные файлы. Очевидный способ мониторинга изменений тома — его сканирование с записью состояния файлов и каталогов и анализ отличий при следующем сканировании. Однако этот процесс может негативно повлиять на производительность системы — особенно на компьютерах, хранящих тысячи и десятки тысяч файлов.

Альтернативный подход для приложения заключается в том, чтобы зарегистрироваться на получение уведомлений об изменении содержимого каталогов. Для этого предназначена Windows-функция *FindFirstChangeNotification* или *ReadDirectoryChangesW*. В качестве входного параметра приложение указывает имя нужного каталога, и функция сообщает о любом изменении в содержимом этого каталога. Хотя этот подход более эффективен, чем сканирование тома, он требует непрерывной работы приложения. При этом приложениям все равно может понадобиться сканирование каталогов, так как *FindFirstChangeNotification* сообщает лишь о факте изменений, а не о конкретных изменениях. В то же время *ReadDirectoryChangesW* принимает от приложения буфер, который FSD заполняет записями об изменениях. Но при переполнении буфера приложение должно быть готово вернуться к сканированию каталога.

NTFS предусматривает третий подход, в котором преодолены недостатки первых двух: приложение может настроить журнал изменений NTFS с помощью функции *DeviceIoControl* и управляющего кода FSCTL_CREATE_USN_JOURNAL; тогда NTFS будет регистрировать информацию об изменениях файлов и каталогов во внутреннем файле — *журнале изменений* (change journal). Этот журнал обычно достаточно велик, что дает приложению шанс обработать все без исключения изменения. Для чтения записей в журнале изменений предназначен управляющий код FSCTL_QUERY_USN_JOURNAL; при этом можно указать, чтобы функция *DeviceIoControl* не завершалась до тех пор, пока в журнале не появятся новые записи.

Квоты томов, индивидуальные для каждого пользователя

Системным администраторам часто бывает нужно отслеживать или ограничивать дисковое пространство, занимаемое пользователями на общих томах в сети. Поэтому NTFS поддерживает управление дисковым пространством на основе квот, позволяя выделять квоты каждому пользователю. NTFS можно настроить на запись в системный журнал события, возникающего в тот момент, когда пользователь превышает пороговое значение, близкое к лимиту. При попытке пользователя занять больше места, чем разрешает его квота, NTFS также регистрирует соответствующее событие в системном журна-

ле и, кроме того, завершает файловый ввод-вывод приложения, вызвавшего нарушение квоты, с кодом ошибки «disk full» («диск заполнен»).

NTFS отслеживает использование тома благодаря тому факту, что помечает файлы и каталоги идентификаторами защиты (SID) пользователей, создавших эти объекты на томе (определение SID см. в главе 8). Сумма логических размеров файлов и каталогов, принадлежащих пользователю, сравнивается с квотой, определенной администратором. Поэтому пользователь не может превысить свою квоту, создав пустой разреженный файл и потом заполняя его ненулевыми значениями. Кстати, хотя 50-килобайтный файл может быть сжат до 10 Кб, при учете используется его исходный размер — 50 Кб.

По умолчанию отслеживание квот на томах отключено. Чтобы разрешить отслеживание квот, указать пороговые значения для выдачи предупреждений, задать ограничения и настроить реакцию NTFS на достижение одного из этих пороговых значений, используйте вкладку Quota (Квота) окна свойств тома (рис. 12-18). Диалоговое окно Quota Entries (Записи квот), которое можно открыть с этой вкладки, позволяет администратору задавать различные лимиты и поведение NTFS для каждого пользователя. Приложения, которым требуется управление на основе квот NTFS, используют COM-интерфейсы квот, в том числе *IDiskQuotaControl*, *IDiskQuotaUser* и *IDiskQuotaEvents*.

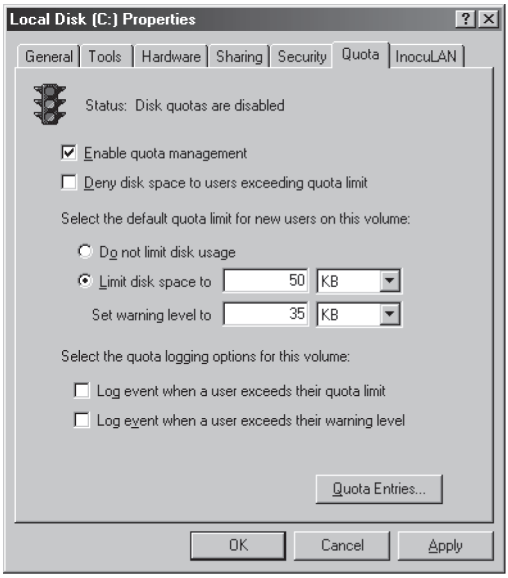


Рис. 12-18. *Окно свойств дискового тома*

Отслеживание ссылок

В пространстве имен оболочки Windows (например, на рабочем столе) можно создавать ярлыки (shortcuts) файлов, находящихся в пространстве имен файловой системы. Такие ярлыки используются, например, в меню Start.

Аналогичным образом OLE-связи позволяют встраивать документы одних приложений в документы, созданные другими приложениями. OLE-связи поддерживаются всеми приложениями из пакета Microsoft Office, включая PowerPoint, Excel и Word.

Хотя OLE-связи дают возможность легко соединять файлы друг с другом и с пространством имен оболочки, управлять ими в прошлом было нелегко. Если пользователь Windows NT 4, Windows 95 или Windows 98 перемещал источник OLE-связи или ярлыка оболочки (источником называется файл или каталог, на который ссылается OLE-связь или ярлык), связь разрывалась, и система предпринимала попытку найти источник связи эвристическим методом. В Windows файловая система NTFS включает поддержку отслеживания распределенных связей (distributed link-tracking), обеспечивающую целостность ярлыков оболочки и OLE-связей при перемещении источников на другой том NTFS в пределах одного домена.

Отслеживание связей в NTFS реализуется на основе необязательного атрибута файла, известного под названием *идентификатор объекта* (object ID). Приложение может назначить такой идентификатор файлу с помощью управляющих кодов файловой системы FSCTL_CREATE_OR_GET_OBJECT_ID (назначает идентификатор, если он еще не назначен) и FSCTL_SET_OBJECT_ID. Идентификаторы объектов можно запросить с помощью управляющих кодов FSCTL_CREATE_OR_GET_OBJECT_ID и FSCTL_GET_OBJECT_ID. Код FSCTL_DELETE_OBJECT_ID позволяет удалять идентификаторы объектов из файлов.

Шифрование

Корпоративные пользователи часто хранят на своих компьютерах конфиденциальную информацию. Хотя данные на серверах компаний обычно надежно защищены, информация, хранящаяся на портативном компьютере, может попасть в чужие руки в случае потери или кражи компьютера. Права доступа к файлам NTFS в таком случае не защитят данные, поскольку полный доступ к томам NTFS можно получить независимо от их защиты — достаточно воспользоваться программами, умеющими читать файлы NTFS вне среды Windows. Более того, права доступа к файлам NTFS становятся бесполезны при использовании другой системы Windows и учетной записи администратора. Вспомните из главы 8, что учетная запись администратора обладает привилегиями захвата во владение и резервного копирования, любая из которых позволяет получить доступ к любому защищенному объекту в обход его параметров защиты.

NTFS поддерживает механизм Encrypting File System (EFS), с помощью которого пользователи могут шифровать конфиденциальные данные. EFS, как и механизм сжатия файлов, полностью прозрачен для приложений. Это означает, что данные автоматически расшифровываются при чтении их приложением, работающим под учетной записью пользователя, который имеет права на просмотр этих данных, и автоматически шифруются при изменении их авторизованным приложением.

ПРИМЕЧАНИЕ NTFS не допускает шифрования файлов, расположенных в корневом каталоге системного тома или в каталоге \Windows, поскольку многие находящиеся там файлы нужны в процессе загрузки, когда EFS еще не активна.

EFS использует криптографические сервисы, предоставляемые Windows в пользовательском режиме, и состоит из драйвера устройства режима ядра, тесно интегрированного с NTFS и DLL-модулями пользовательского режима, которые взаимодействуют с подсистемой локальной аутентификации (LSASS) и криптографическими DLL.

Доступ к зашифрованным файлам можно получить только с помощью закрытого ключа из криптографической пары EFS (которая состоит из закрытого и открытого ключей), а закрытые ключи защищены паролем учетной записи. Таким образом, без пароля учетной записи, авторизованной для просмотра данных, доступ к зашифрованным EFS файлам на потерянных или краденых портативных компьютерах нельзя получить никакими средствами (кроме грубого перебора паролей).

Windows-функции *EncryptFile* и *DecryptFile* позволяют шифровать и дешифровать файлы, а *FileEncryptionStatus* — получать атрибуты файла или каталога, связанного с EFS, — например, чтобы определить, зашифрован ли данный файл или каталог.

Поддержка POSIX

Как говорилось в главе 2, одно из требований к Windows состояло в том, что она должна полностью поддерживать стандарт POSIX 1003.1. Стандарт POSIX требует от файловой системы поддержки имен файлов и каталогов, чувствительных к регистру букв, цепочечных разрешений (traversal permissions) (для доступа к файлу нужны права на доступ к каждому каталогу на пути к этому файлу), метки времени изменения файла (отличной от метки времени последней модификации файла в MS-DOS) и жестких связей. Вся эта функциональность в NTFS реализована.

Дефрагментация

Многие верят в то, что NTFS якобы автоматически оптимизирует размещение файлов на диске, не допуская их фрагментации. Хотя она стремится записывать файлы в непрерывные области, со временем файлы на томе могут стать фрагментированными — особенно когда свободного пространства мало. Файл является фрагментированным, если его данные занимают несмежные кластеры. Так, на рис. 12-19 показан фрагментированный файл, состоящий из трех фрагментов. Но, как и большинство файловых систем (включая версию FAT в Windows), NTFS не предпринимает специальных мер по поддержанию непрерывного размещения файлов на диске — разве что резервирует область дискового пространства, называемую зоной главной таблицы файлов (master file table, MFT), где и хранится MFT. (NTFS разрешает выделять место под файлы из зоны MFT, когда на томе остается мало сво-

бодного пространства.) Подробнее о MFT см. раздел «Главная таблица файлов» далее в этой главе.

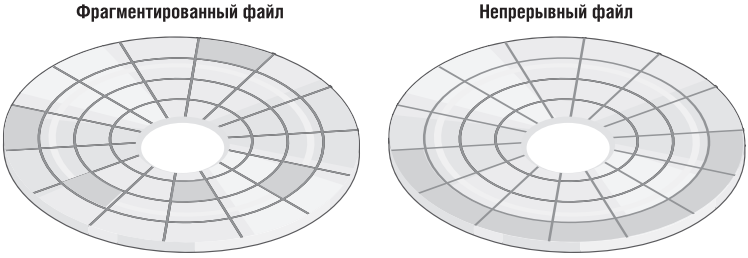


Рис. 12-19. *Фрагментированный и непрерывный файлы*

Для упрощения разработки независимых средств дефрагментации дисков в Windows включен API дефрагментации, с помощью которого подобные утилиты могут перемещать файловые данные так, чтобы файлы занимали непрерывные кластеры. Этот API реализован на основе управляющих кодов файловой системы: FSCTL_GET_VOLUME_BITMAP (возвращает карту свободных и занятых кластеров тома), FSCTL_GET_RETRIEVAL_POINTERS (возвращает карту кластеров, занятых файлом) и FSCTL_MOVE_FILE (перемещает файл).

В Windows имеется встроенная программа дефрагментации, доступная через утилиту Disk Defragmenter (\Windows\System32\Dfrg.msc). Ей присущ ряд ограничений, в частности в Windows 2000 эту утилиту нельзя запускать из командной строки или автоматически запускать по заданному расписанию. В Windows XP и Windows Server 2003 у программы дефрагментации появился интерфейс командной строки, \Windows\System32\Defrag.exe, позволяющий запускать процесс дефрагментации интерактивно или по расписанию, но она по-прежнему не сообщает детальных отчетов и не поддерживает такие возможности, как исключение определенных файлов или каталогов из процесса дефрагментации. Сторонние утилиты дефрагментации дисков, как правило, предлагают более богатую функциональность.

В Windows XP поддержка дефрагментации для NTFS была переписана, чтобы снять часть ограничений, которые были в Windows 2000. Так, в Windows 2000 поддержка дефрагментации NTFS опиралась на диспетчер кэша, что создавало ряд ограничений, например невозможность перемещения файловых блоков размером более 256 Кб или дефрагментации файлов метаданных NTFS. (Заметьте, что 256 Кб — это размер представлений, поддерживаемых диспетчером кэша. Подробнее о диспетчере кэша см. главу 11.) В реализации дефрагментации NTFS в Windows XP и Windows Server 2003 существует лишь одно ограничение — нельзя дефрагментировать страничные файлы и файлы журналов NTFS.

Поддержка доступа только для чтения

До Windows XP драйвер файловой системы NTFS поддерживал монтирование тома исключительно на записываемом носителе, куда он должен был

помещать файлы журналов транзакций. Драйверы NTFS в Windows XP и Windows Server 2003 могут монтировать тома на носителях только для чтения; эта функциональность нужна во встраиваемых системах, в которых базовые образы файловой системы (base file system images) доступны только для чтения.

Драйвер файловой системы NTFS

Как отмечалось в главе 9, подсистема ввода-вывода Windows устроена так, что NTFS и другие файловые системы представляют собой загружаемые драйверы устройств режима ядра. Они неявно вызываются приложениями, использующими Windows или другие API ввода-вывода (например, POSIX). Как показано на рис. 12-20, подсистемы окружения вызывают системные сервисы, которые в свою очередь находят соответствующие загруженные драйверы и вызывают их. (О диспетчеризации системных сервисов см. раздел «Диспетчеризация системных сервисов» главы 3.)

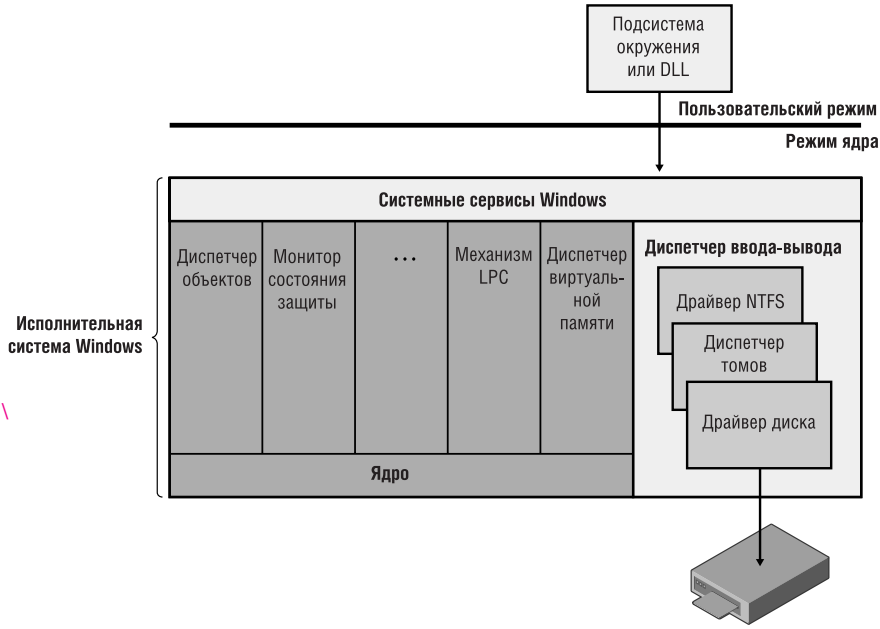


Рис. 12-20. *Компоненты подсистемы ввода-вывода в Windows*

Драйверы передают друг другу запросы ввода-вывода, вызывая диспетчер ввода-вывода исполнительной системы. Использование диспетчера ввода-вывода в качестве промежуточного звена обеспечивает независимость каждого драйвера, что позволяет загружать и выгружать его без последствий для других драйверов. Кроме того, драйвер NTFS взаимодействует с тремя другими компонентами исполнительной системы (рис. 12-21), тесно связанными с файловыми системами.

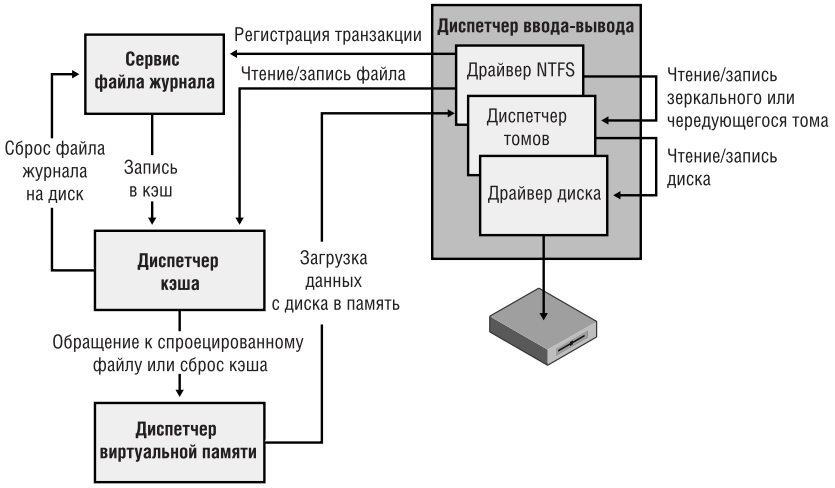


Рис. 12-21. *NTFS и связанные с ней компоненты*

Сервис файла журнала (log file service, LFS) является частью NTFS и предоставляет функции для поддержки журнала изменений на диске. Файл журнала LFS используется при восстановлении тома NTFS в случае аварии системы (подробнее о LFS см. раздел «Сервис файла журнала» далее в этой главе).

Диспетчер кэша — компонент исполнительной системы, предоставляющий общесистемные сервисы кэширования для NTFS и драйверов других файловых систем, в том числе сетевых (т. е. для серверов и редиректоров). Все файловые системы, реализованные в Windows, получают доступ к кэшированным файлам, проецируя их на системное адресное пространство, а затем считывая соответствующие участки виртуальной памяти. С этой целью диспетчер кэша предоставляет диспетчеру памяти специализированный интерфейс файловых систем. Когда программа пытается обратиться к какой-либо части файла, не загруженной в кэш (промах кэша), диспетчер памяти вызывает NTFS для обращения к драйверу диска и загрузки нужных файловых данных. Диспетчер кэша оптимизирует дисковый ввод-вывод, вызывая диспетчер памяти (для сброса на диск содержимого кэша в фоновом режиме) из потоков подсистемы отложенной записи.

NTFS участвует в модели объектов Windows, реализуя файлы в виде объектов. Такая реализация допускает совместное использование файлов и их защиту диспетчером объектов, который управляет всеми объектами уровня исполнительной системы (о диспетчере объектов см. главу 3).

Приложение создает файлы и обращается к ним так же, как и к любым другим объектам Windows, — через описатели объектов. К тому времени, когда запрос ввода-вывода достигает NTFS, диспетчер объектов и система защиты уже убедились в наличии у вызывающего процесса прав на запрошенные им виды доступа к объекту «файл». Система защиты сравнила маркер доступа вызывающего процесса с элементами ACL этого объекта «файл» (подробнее об ACL см. главу 8), а диспетчер ввода-вывода преобразовал опи-

сатель файла в указатель на объект «файл». NTFS использует информацию из объекта «файл» для обращения к файлу на диске.

На рис. 12-22 показаны структуры данных, связывающие описатель файла со структурой файловой системы на диске.

NTFS получает адрес файла на диске из объекта «файл» по нескольким указателям. Как видно из рис. 12-22, объект «файл», представляющий один вызов системного сервиса для открытия файла, указывает на *блок управления потоком данных* (stream control block, SCB) для атрибута, который вызывающая программа пытается считать или записать. В нашем случае процесс открыл как безымянный атрибут данных, так и именованный поток (альтернативный атрибут данных) файла. SCB представляют отдельные атрибуты файла и содержат информацию о том, как искать конкретные атрибуты внутри файла. Все SCB файла указывают на общую структуру данных — *блок управления файлом* (file control block, FCB). FCB содержит указатель на запись файла в главной таблице файлов (MFT), о которой мы поговорим в следующем разделе.

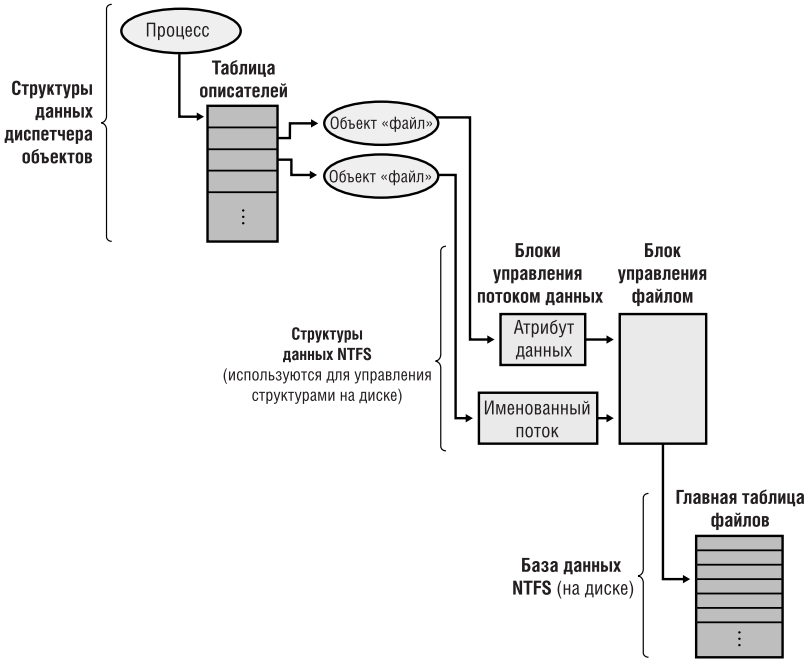


Рис. 12-22. *Структуры данных NTFS*

Структура NTFS на диске

Здесь мы рассмотрим структуру тома NTFS, включая способы разбиения дискового пространства и его организации в кластеры, принципы хранения на диске реальных файловых данных и информации об атрибутах, а также поясним, как работает механизм сжатия данных в NTFS.

Тома

Структура NTFS начинается с тома. *Том* (volume) соответствует логическому разделу на диске и создается при форматировании диска или его части под NTFS. Оснастка Disk Management (Управление дисками) консоли MMC также позволяет создать том RAID, охватывающий несколько дисков.

На диске может быть один или несколько томов. NTFS обрабатывает каждый том независимо от других. Три примера конфигурации 150-мегабайтного жесткого диска показаны на рис. 12-23.

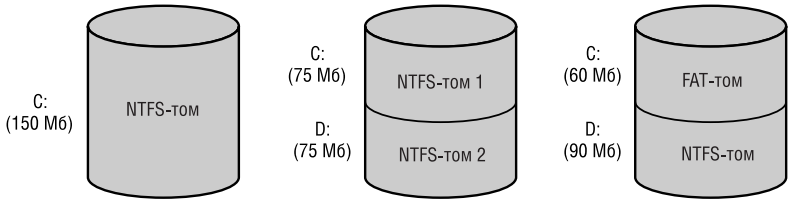


Рис. 12-23. *Примеры конфигураций диска*

Том состоит из набора файлов и свободного пространства, оставшегося в данном разделе диска. В FAT том также содержит области, специально отформатированные для использования файловой системой. Но в томе NTFS все данные файловой системы вроде битовых карт, каталогов и даже начального загрузочного кода хранятся как обычные файлы.

ПРИМЕЧАНИЕ У NTFS-томов в Windows 2000 дисковый формат версии 3.0; в Windows XP и Windows Server 2003 в этот формат были внесены незначительные изменения, и теперь используется его версия 3.1. Номер версии тома хранится в файле метаданных $Volume.

Кластеры

Размер кластера на томе NTFS, или *кластерный множитель* (cluster factor), устанавливается при форматировании тома командой *format* или в оснастке Disk Management (Управление дисками). Размер кластера по умолчанию определяется размером тома, но всегда содержит целое число физических секторов с дискретностью N^2 (т. е. 1 сектор, 2 сектора, 4 сектора, 8 секторов и т. д.). Кластерный множитель выражается числом байтов в кластере, например 512 байт, 1 Кб или 2 Кб.

Внутренне NTFS работает только с кластерами. (Однако NTFS инициирует низкоуровневые операции ввода-вывода на томе, выравнивая передаваемые

данные по размеру сектора и подгоняя их объем под значение, кратное размеру секторов.) NTFS использует кластер как единицу выделения пространства для поддержания независимости от размера физического сектора. Это позволяет NTFS эффективно работать с очень большими дисками, используя кластеры большего размера, и поддерживать нестандартные диски с размером секторов, отличным от 512 байтов. Применение бóльших кластеров на бóльших томах уменьшает фрагментацию и ускоряет выделение свободного пространства за счет небольшого проигрыша в эффективности использования дискового пространства. Команда *format* или оснастка Disk Management выбирает кластерный множитель в зависимости от размера тома, но это значение можно изменить.

NTFS адресуется к конкретным местам на диске, используя *логические номера кластеров* (logical cluster numbers, LCN). Для этого все кластеры на томе просто нумеруются по порядку — от начала до конца. Для преобразования LCN в физический адрес на диске NTFS умножает LCN на кластерный множитель и получает байтовое смещение от начала тома, воспринимаемое интерфейсом драйвера диска. На данные внутри файла NTFS ссылается по *виртуальным номерам кластеров* (virtual cluster numbers, VCN), нумеруя кластеры, которые принадлежат конкретному файлу (от 0 до *m*). VCN не обязательно должны быть физически непрерывными.

Главная таблица файлов

В NTFS все данные, хранящиеся на томе, содержатся в файлах. Это относится и к структурам данных, используемым для поиска и выборки файлов, к начальному загрузочному коду и битовой карте, в которой регистрируется состояние пространства всего тома (метаданные NTFS). Хранение всех видов данных в файлах позволяет файловой системе легко находить и поддерживать данные, а каждый файл может быть защищен дескриптором защиты. Кроме того, при появлении плохих секторов на диске, NTFS может переместить файлы метаданных.

Главная таблица файлов (MFT) занимает центральное место в структуре NTFS-тома. MFT реализована как массив записей о файлах. Размер каждой записи фиксирован и равен 1 Кб (см. раздел «Записи о файлах» далее в этой главе). Логически MFT содержит по одной строке на каждый файл тома, включая строку для самой MFT. Кроме MFT, на каждом томе NTFS имеется набор файлов метаданных с информацией, необходимой для реализации структуры файловой системы. Имена всех файлов метаданных NTFS начинаются со знака доллара ($), хотя эти знаки скрыты. Так, имя файла MFT — $Mft. Остальные файлы NTFS-тома являются обычными файлами и каталогами (рис. 12-24).

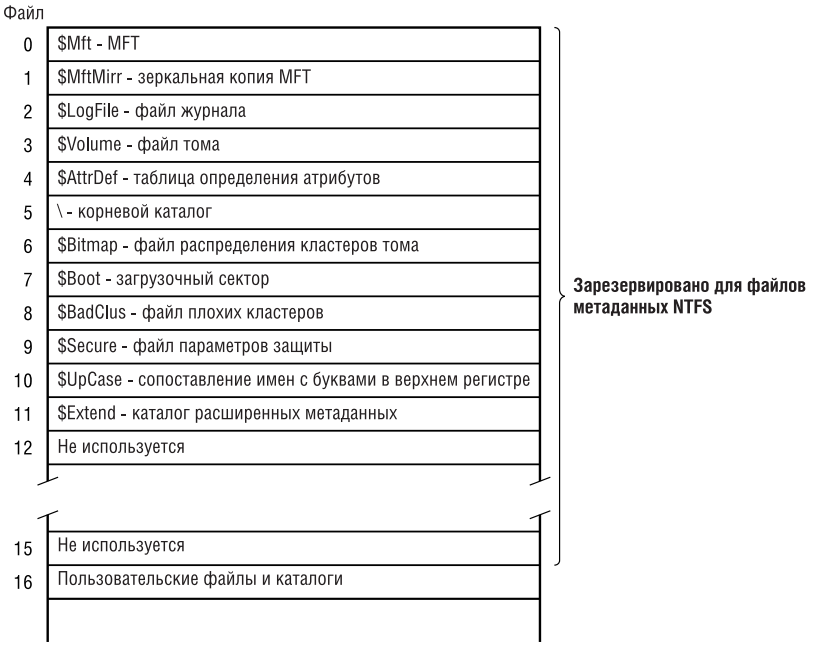


Рис. 12-24. *Записи MFT для файлов метаданных NTFS*

Обычно каждая запись MFT соответствует отдельному файлу. Но если у файла много атрибутов или он сильно фрагментирован, для него может понадобиться более одной записи. Тогда первая запись MFT, хранящая адреса других записей, называется *базовой* (base file record).

ЭКСПЕРИМЕНТ: просмотр MFT

Утилита Nfi из OEM Support Tools (входит в отладочные средства Windows; ее можно скачать с *support.microsoft.com/support/kb/articles/Q253/0/66.asp*) позволяет получить дамп содержимого MFT тома NTFS и преобразовать номер кластера тома или номер сектора физического диска (не для RAID-томов) в имя соответствующего файла (если кластер или сектор занят файлом). Первые 16 элементов MFT зарезервированы для файлов метаданных, но записи для файлов необязательных метаданных (которые присутствуют только при использовании на томе соответствующих возможностей) находятся вне этой области: \$Extend\$Quota, \$Extend\$ObjId, \$Extend\$UsnJrnl и \$Extend\$Reparse. Следующий дамп получен для тома, на котором используются точки повторного разбора ($Reparse), квоты ($Quota) и идентификаторы объектов ($ObjId).

```
C:\>nfi G:\
NTFS File Sector Information Utility.
```

см. след. стр.

```
Copyright (C) Microsoft Corporation 1999. All rights reserved.

File 0
Master File Table ($Mft)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 32-52447 (0x20-0xccdf)
    $BITMAP (nonresident)
        logical sectors 16-23 (0x10-0x17)

File 1
Master File Table Mirror ($MftMirr)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 2048728-2048735 (0x1f42d8-0x1f42df)

File 2
Log File ($LogFile)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 2048736-2073343 (0x1f42e0-0x1fa2ff)
File 3
DASD ($Volume)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $OBJECT_ID (resident)
    $SECURITY_DESCRIPTOR (resident)
    $VOLUME_NAME (resident)
    $VOLUME_INFORMATION (resident)
    $DATA (resident)

File 4
Attribute Definition Table ($AttrDef)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (nonresident)
        logical sectors 512256-512263 (0x7d100-0x7d107)

File 5
Root Directory
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $SECURITY_DESCRIPTOR (resident)
    $INDEX_ROOT $I30 (resident)
    $INDEX_ALLOCATION $I30 (nonresident)
```

```
        logical sectors 2073416-2073423 (0x1fa348-0x1fa34f)
    $BITMAP $I30 (resident)

File 6
Volume Bitmap ($BitMap)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 2073424-2073675 (0x1fa350-0x1fa44b)

File 7
Boot Sectors ($Boot)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (nonresident)
        logical sectors 0-15 (0x0-0xf)
File 8
Bad Cluster List ($BadClus)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (resident)
    $DATA $Bad (nonresident)

File 9
Security ($Secure)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA $SDS (nonresident)
        logical sectors 2073932-2074447 (0x1fa54c-0x1fa74f)
        logical sectors 523160-523163 (0x7fb98-0x7fb9b)
    $INDEX_ROOT $SDH (resident)
    $INDEX_ROOT $SII (resident)
    $INDEX_ALLOCATION $SDH (nonresident)
        logical sectors 1876152-1876159 (0x1ca0b8-0x1ca0bf)
    $INDEX_ALLOCATION $SII (nonresident)
        logical sectors 24-31 (0x18-0x1f)
    $BITMAP $SDH (resident)
    $BITMAP $SII (resident)

File 10
Upcase Table ($UpCase)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 2073676-2073931 (0x1fa44c-0x1fa54b)
```

см. след. стр.

```
File 11
Extend Table ($Extend)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $INDEX_ROOT $I30 (resident)

File 12
(unknown/unnamed)
    $STANDARD_INFORMATION (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (resident)
File 13
(unknown/unnamed)
    $STANDARD_INFORMATION (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (resident)

File 14
(unknown/unnamed)
    $STANDARD_INFORMATION (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (resident)

File 15
(unknown/unnamed)
    $STANDARD_INFORMATION (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (resident)

File 24
\$Extend\$Quota
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $INDEX_ROOT $O (resident)
    $INDEX_ROOT $Q (resident)

File 25
\$Extend\$ObjId
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $INDEX_ROOT $O (resident)

File 26
\$Extend\$Reparse
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $INDEX_ROOT $R (resident)
```

При первом обращении к тому NTFS должна смонтировать его, т. е. считать с диска метаданные и сформировать внутренние структуры данных, необходимые для обработки обращений к файловой системе. Чтобы смонтировать том, NTFS ищет в загрузочном секторе физический адрес MFT на диске. Запись о самой MFT является первым элементом в этой таблице, вторая запись указывает на файл в середине диска ($MftMirr), который называется зеркальной копией MFT и содержит копию первых нескольких записей MFT. Если по каким-либо причинам считать часть MFT не удастся, для поиска файлов метаданных будет использована именно эта копия MFT.

Найдя запись для MFT, NTFS получает из ее атрибута данных информацию о сопоставлении VCN и LCN и сохраняет ее в памяти. В каждой группе (run) хранится сопоставление VCN-LCN и длина этой группы — вот и вся информация, необходимая для того, чтобы найти LCN по VCN. Эта информация сообщает NTFS, где на диске искать группы, образующие MFT. (О группах см. раздел «Резидентные и нерезидентные атрибуты» далее в этой главе.) Затем NTFS обрабатывает записи MFT еще для нескольких файлов метаданных и открывает эти файлы. Наконец, NTFS выполняет операцию восстановления файловой системы (см. раздел «Восстановление» далее в этой главе) и открывает остальные файлы метаданных. С этого момента пользователь может обращаться к данному дисковому тому.

ПРИМЕЧАНИЕ Для упрощения восприятия материала в тексте и на схемах в этой главе группа обозначается как сущность, содержащая VCN, LCN и длину группы. Но на самом деле NTFS сжимает эту информацию на диске в пары «LCN — следующий VCN». Зная начальный VCN, NTFS может определить длину группы простым вычитанием начального VCN из следующего VCN.

В процессе работы системы NTFS ведет запись в другой важный файл метаданных — *файл журнала* с именем $LogFile. NTFS использует его для регистрации всех операций, влияющих на структуру тома NTFS, в том числе для регистрации создания файлов и выполнения любых команд вроде *Copy*, модифицирующих структуру каталогов. Файл журнала используется и при восстановлении тома NTFS после аварии системы.

Еще один элемент MFT зарезервирован для корневого каталога (также обозначаемого как «\»). Его запись содержит индекс файлов и каталогов, хранящихся в корне структуры каталогов NTFS. Когда NTFS впервые получает запрос на открытие файла, она начинает его поиск с записи корневого каталога. Открыв файл, NTFS сохраняет файловую ссылку MFT для этого файла и поэтому в следующий раз, когда понадобится считать или записать тот же файл, сможет напрямую обратиться к его записи в MFT.

NTFS регистрирует распределение дискового пространства в *файле битовой карты* (bitmap file) с именем $Bitmap. Атрибут данных для файла битовой карты содержит битовую карту, каждый бит которой представляет кластер тома и сообщает, свободен кластер или выделен.

Файл защиты (security file) с именем $Secure хранит базу данных дескрипторов защиты, действующих в пределах тома. Дескрипторы защиты файлов и каталогов NTFS можно настраивать индивидуально, но для экономии места NTFS хранит настройки дескрипторов защиты в общем файле, который позволяет файлам и каталогам с одинаковыми параметрами защиты ссылаться на один и тот же дескриптор защиты. Такая оптимизация дает существенную экономию в большинстве сред, потому что в них целые деревья каталогов имеют одинаковые параметры защиты.

Другой системный файл, *загрузочный* (boot file), с именем $Boot хранит код начальной загрузки Windows. Для успешного запуска системы код начальной загрузки должен находиться на диске в определенном месте. При форматировании команда *format* определяет это место в виде файла, создавая для него запись в MFT. При этом NTFS следует своим правилам, согласно которым все данные хранятся на диске в виде файлов. Загрузочный файл, как и файлы метаданных NTFS, может быть защищен индивидуальным дескриптором защиты. В такой модели «на диске есть только файлы» код начальной загрузки может быть изменен путем обычного файлового ввода-вывода, хотя загрузочный файл защищен от редактирования.

NTFS поддерживает *файл плохих кластеров* (bad-cluster file) с именем $BadClus, в котором регистрируются все сбойные участки дискового тома, и *файл тома* (volume file) с именем $Volume, который содержит имя тома, версию NTFS, под которую отформатирован том, и бит, устанавливаемый при каком-либо повреждении диска. Если этот бит установлен, диск должен быть восстановлен утилитой Chkdsk. *Файл сопоставления имен с буквами в верхнем регистре* (uppercase file) с именем $UpCase включает таблицу трансляции букв между верхним и нижним регистрами. NTFS также поддерживает файл, содержащий *таблицу определения атрибутов* (attribute definition table), с именем $AttrDef; в нем определяются типы атрибутов, поддерживаемые томом, и указывается, являются ли они индексируемыми, следует ли их восстанавливать в ходе операции восстановления системы и т. д.

Некоторые файлы метаданных NTFS хранит в каталоге расширенных метаданных $Extend, в том числе помещая туда *файл идентификаторов объектов* ($ObjId), *файл квот* ($Quota), *файл журнала регистрации изменений* ($UsnJrnl) и *файл точек повторного разбора* ($Reparse). В этих файлах содержится информация, относящаяся к дополнительным возможностям NTFS. Файл идентификаторов объектов хранит идентификаторы объектов «файл», файл квот — значения квот и информацию о поведении томов, на которых используются квоты, файл точек повторного разбора — список файлов и каталогов, включающих данные точек повторного разбора, а в файле журнала изменений регистрируются изменения файлов и каталогов.

ЭКСПЕРИМЕНТ: просмотр информации NTFS

Для просмотра информации о NTFS-томе, в том числе о размещении и размере MFT и зоны MFT, вы можете использовать в Windows 2000 утилиту NTFSInfo (с сайта *www.sysinternals.com*), а в Windows XP или Windows Server 2003 — встроенную программу командной строки Fsutil.exe:

```
C:\Windows\System32>fsutil fsinfo ntfsinfo c:
NTFS Volume Serial Number :       0xe82828e72828b68a
Version :                         3.1
Number Sectors :                  0x0000000001e461b7
Total Clusters :                  0x00000000003c8c36
Free Clusters  :                  0x00000000000164c8
Total Reserved :                  0x00000000000001b0
Bytes Per Sector  :               512
Bytes Per Cluster :               4096
Bytes Per FileRecord Segment    : 1024
Clusters Per FileRecord Segment : 0
Mft Valid Data Length :           0x0000000006413800
Mft Start Lcn  :                  0x00000000000c5294
Mft2 Start Lcn :                  0x000000000002f427
Mft Zone Start :                  0x00000000003bf7e0
Mft Zone End   :                  0x00000000003bf800
```

Структура файловых ссылок

Файл на томе NTFS идентифицируется 64-битным значением, которое называется *файловой ссылкой* (file reference). Файловая ссылка состоит из номера файла и номера последовательности. Номер файла равен позиции его записи в MFT минус 1 (или позиции базовой записи в MFT минус 1, если файл требует несколько записей). Номер последовательности в файловой ссылке увеличивается на 1 при каждом повторном использовании позиции записи в MFT, что позволяет NTFS проверять внутреннюю целостность файловой системы. Файловую ссылку иллюстрирует рис. 12-25.

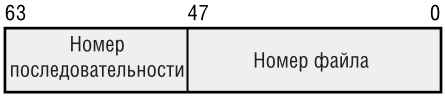


Рис. 12-25. *Файловая ссылка*

Записи о файлах

NTFS рассматривает файл не просто как хранилище текстовых или двоичных данных, а как совокупность пар атрибутов и их значений, одна из которых содержит данные файла (соответствующий атрибут называется *неименованным атрибутом данных*). Другие атрибуты включают имя файла,

метку времени и, возможно, дополнительные именованные атрибуты данных. Запись MFT для небольшого файла показана на рис. 12-26.

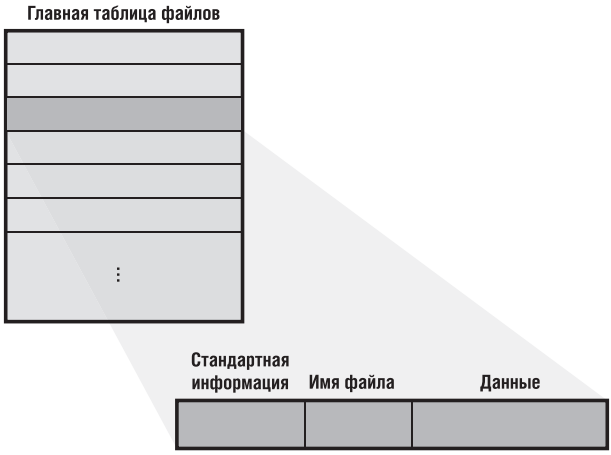


Рис. 12-26. *Запись MFT для небольшого файла*

Каждый атрибут файла хранится в файле как отдельный поток байтов. Строго говоря, NTFS читает и записывает не файлы, а потоки атрибутов. NTFS поддерживает следующие операции над атрибутами: создание, удаление, чтение (как диапазон байтов) и запись (как диапазон байтов). Сервисы чтения и записи обычно имеют дело с неименованным атрибутом данных. Однако вызывающая программа может указать другой атрибут данных, используя синтаксис именованных потоков данных.

В таблице 12-4 перечислены атрибуты для файлов на томах NTFS (не у каждого файла есть все эти атрибуты).

Таблица 12-4. *Атрибуты файлов в NTFS*

Атрибут	Имя атрибута	Описание
Информация о томе	$VOLUME_ INFORMATION, $VOLUME_NAME	Эти атрибуты имеются только в файле метаданных $Volume. Они хранят версию и метку тома
Стандартная информация	$STANDARD_ INFORMATION	Атрибуты файла «только для чтения», «архивный» и др., метки времени создания и последней модификации, число каталогов, ссылающихся на данный файл (счетчик жестких связей)
Имя файла	$FILE_NAME	Имя файла в Unicode-символах. У файла может быть несколько атрибутов имени, например, если существует жесткая связь с данным файлом или если для его длинного имени автоматически сгенерировано краткое имя, используемое программами MS-DOS и 16-разрядной Windows

Таблица 12-4. (*окончание*)

Атрибут	Имя атрибута	Описание
Дескриптор защиты	$SECURITY_DESCRIPTOR	Этот атрибут обеспечивает обратную совместимость с прежними версиями NTFS. Версия NTFS, реализованная в Windows, хранит все дескрипторы защиты в файле метаданных $Secure для совместного использования файлами и каталогами с одинаковыми параметрами защиты. Прежние версии NTFS хранили закрытую информацию о дескрипторе защиты в каждом файле и каталоге
Данные	$DATA	Содержимое файла. В NTFS файл имеет один неименованный атрибут данных, и, возможно, дополнительные именованные атрибуты данных, т. е. в файле может быть несколько потоков данных. В каталоге нет атрибута данных по умолчанию, но могут присутствовать необязательные именованные атрибуты данных
Корень индекса, выделенная группа индексов и битовая карта индексов	$INDEX_ROOT, $INDEX_ALLOCATION, $BITMAP	Эти атрибуты используются для выделения места под имена файлов и создания битовой карты индексов для больших каталогов (только каталогов)
Список атрибутов	$ATTRIBUTE_LIST	Список атрибутов, составляющих файл, и файловые ссылки на запись MFT, в которой находятся все атрибуты. Этот редко используемый атрибут присутствует, когда файл требует более одной записи MFT
Идентификатор объекта	$OBJECT_ID	64-байтовый идентификатор файла или каталога, в котором младшие 16 байтов (128 битов) уникальны для тома. Сервисы отслеживания связей назначают идентификаторы объектов ярлыкам оболочки и файлам—источникам OLE-связей. NTFS предоставляет API-функции, позволяющие открывать файлы и каталоги не по именам, а по идентификаторам
Информация повторного разбора	$REPARSE_POINT	Этот атрибут хранит данные точки повторного разбора, сопоставленной с файлом; присутствует в точках соединения и монтирования
Расширенные атрибуты	$EA, $EA_INFORMATION	Расширенные атрибуты, иногда используемые для обратной совместимости с приложениями OS/2
Информация EFS	$EFS	В этом атрибуте EFS хранит данные, используемые для управления шифрованием файла, например шифрованную версию ключа, необходимого для расшифровки файла, и список пользователей, имеющих право на доступ к этому файлу

В таблице 12-4 даны имена атрибутов, но на самом деле атрибуты соответствуют числовым кодам типов, используемым NTFS для упорядочения атрибутов в записи о файле. Файловые атрибуты в записи MFT размещаются в порядке возрастания числовых значений этих кодов. Некоторые типы атрибутов встречаются в записи дважды — например, если у файла несколько атрибутов данных или несколько имен.

Каждый атрибут в записи о файле идентифицируется кодом типа атрибута, имеет значение и необязательное имя. Значение атрибута представляет собой байтовый поток. Так, значением атрибута $FILE_NAME является имя файла, значением атрибута $DATA — произвольный набор байтов, сохраненный пользователем в файле.

У большинства атрибутов нет имен, хотя у $DATA и атрибутов, связанных с индексом, они обычно есть. Имена позволяют различать атрибуты файла, относящиеся к одному типу. Например, в файле с именованным потоком данных есть два атрибута $DATA: неименованный атрибут $DATA, хранящий неименованный по умолчанию поток данных, и именованный атрибут $DATA с именем дополнительного потока данных.

Имена файлов

NTFS и FAT допускают длину каждого имени файла в пути до 255 символов. Эти имена могут содержать Unicode-символы, точки и пробелы. Однако длина имен файлов в FAT, встроенной в MS-DOS, ограничена 8 символами (не-Unicode), за которыми следует расширение из трех символов, отделенное точкой. Рис. 12-27 иллюстрирует различные пространства имен файлов, поддерживаемые Windows, и показывает, как они перекрываются.

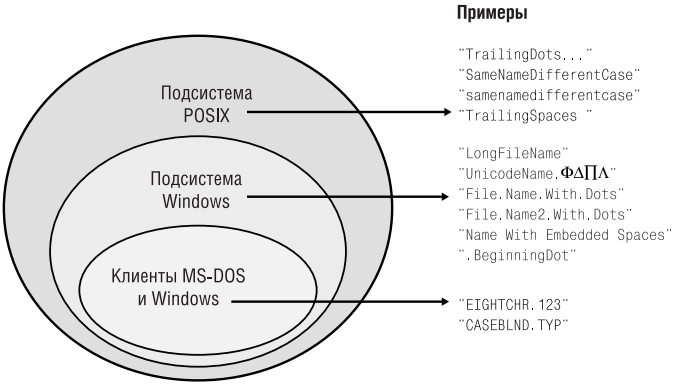


Рис. 12-27. *Пространства имен файлов, поддерживаемые Windows*

POSIX требует самого большого пространства имен из всех подсистем, поддерживаемых Windows. Поэтому пространство имен NTFS эквивалентно пространству имен POSIX. Подсистема POSIX может создавать имена, невидимые приложениям Windows и MS-DOS, в том числе имена с концевыми точками и концевыми пробелами. Создание файла в большом пространстве

имен POSIX обычно не является проблемой, потому что вы создаете такой файл для использования подсистемой POSIX или ее клиентской системой.

Но взаимосвязь между 32-разрядными Windows-приложениями и программами MS-DOS и 16-разрядной Windows намного теснее. Область, отведенная Windows на рис. 12-26, представляет имена файлов, которые подсистема Windows может создавать на томе NTFS, хотя такие имена невидимы программам MS-DOS и 16-разрядной Windows. В эту группу входят имена файлов, не соответствующие формату «8.3» (длинные имена, имена с Unicode-символами, с несколькими точками или начинающиеся с точки, а также имена с внутренними пробелами). При создании файла с таким именем NTFS автоматически генерирует для него альтернативное имя в стиле MS-DOS. Windows показывает такие имена при использовании команды *dir* с ключом */x*.

Имена MS-DOS — полнофункциональные псевдонимы файлов NTFS и хранятся в том же каталоге, что и длинные имена. На рис. 12-28 показана запись MFT для файла с автоматически сгенерированным MS-DOS-именем.

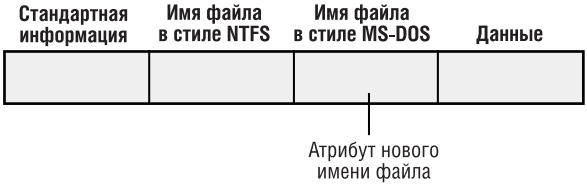


Рис. 12-28. *Запись MFT с атрибутом имени файла для MS-DOS*

Имя NTFS и сгенерированное имя MS-DOS хранятся в той же записи и относятся к одному и тому же файлу. Имя MS-DOS можно использовать для открытия, чтения, записи и копирования файла. Если пользователь переименовывает файл, заменяя длинное имя на краткое или наоборот, новое имя заменяет оба существовавших варианта. Если новое имя является недопустимым для MS-DOS, NTFS генерирует для файла другое MS-DOS-имя.

ПРИМЕЧАНИЕ Аналогичным образом реализуются жесткие связи POSIX. При создании жесткой связи с POSIX-файлом NTFS добавляет в запись MFT дополнительный атрибут имени файла. Однако эти две ситуации имеют одно отличие. Когда пользователь удаляет файл POSIX, у которого было несколько имен (жестких связей), запись о файле и сам файл остаются. Файл и его запись удаляются только после удаления последнего имени (жесткой связи). Если у файла есть и имя NTFS, и автоматически сгенерированное имя MS-DOS, пользователь может удалить файл по любому из этих имен.

Вот алгоритм, применяемый NTFS при генерации краткого MS-DOS-имени из длинного.

1. Удалить из длинного имени все символы, недопустимые в именах MS-DOS, включая пробелы и Unicode-символы. Удалить начальную и концевую точки, а также все внутренние точки, кроме последней.

2. Урезать часть строки перед точкой (если она есть) до шести символов и добавить строку «~n» (где *n* — порядковый номер, который начинается с 1; он нужен, чтобы различать файлы, урезание имен которых дает одинаковый результат). Урезать строку после точки (если она есть) до трех символов.
3. Преобразовать полученный набор символов в верхний регистр. MS-DOS нечувствительна к регистру букв в именах файлов, но эта операция гарантирует, что NTFS не сгенерирует для файла новое имя, отличающееся от старого лишь регистром.
4. Если сгенерированное имя дублирует уже имеющееся в каталоге, увеличить порядковый номер в строке «~n» на 1 (или на большее значение).

В таблице 12-5 показаны длинные имена файлов с рис. 12-26 и их MS-DOS-версии, сгенерированные NTFS. Приведенный выше алгоритм и примеры на рис. 12-26 должны дать вам представление об именах в стиле MS-DOS, генерируемых NTFS.

ПРИМЕЧАНИЕ Вы можете отключить генерацию кратких имен, присвоив параметру реестра HKLM\System\CurrentControlSet\Control\FileSystem\ NtfsDisable8dot3NameCreation значение типа DWORD, равное 1, хотя обычно это не рекомендуется (из-за вероятной несовместимости приложений, которые полагаются на такую функциональность).

Таблица 12-5. *Краткие имена, генерируемые NTFS*

Длинное имя	Краткое имя, сгенерированное NTFS
LongFileName	LONGFI~1
UnicodeName.✦✤✰✯	UNICOD~1
File.Name.With.Dots	FILENA~1.DOT
File.Name2.With.Dots	FILENA~2.DOT
Name With Embedded Spaces	NAMEWI~1
.BeginningDot	BEGINN~1

Резидентные и нерезидентные атрибуты

Если файл невелик, все его атрибуты и их значения (например, файловые данные) умещаются в одной записи файла. Когда значение атрибута хранится непосредственно в MFT, атрибут называется *резидентным* (например, все атрибуты на рис. 12-27 являются резидентными). Некоторые атрибуты всегда резидентны — по ним NTFS находит нерезидентные атрибуты. Так, атрибуты «стандартная информация» и «корень индекса» всегда резидентны.

Каждый атрибут начинается со стандартного заголовка, в котором содержится информация об атрибуте, используемая NTFS для базового управления атрибутами. В заголовке, который всегда является резидентным, регистрируется, резидентно ли значение данного атрибута. В случае резидентных атрибутов заголовок также содержит смещение значения атрибута от начала заголовка и длину этого значения (рис. 12-29).

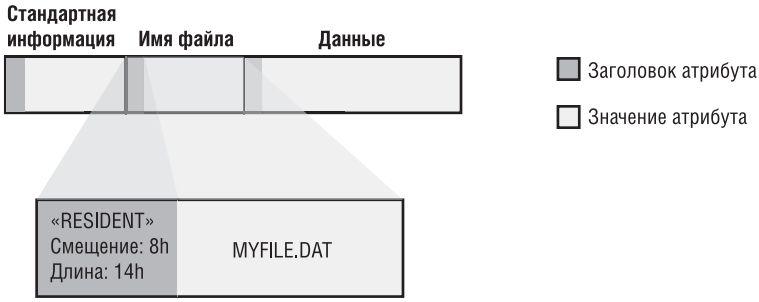


Рис. 12-29. *Заголовок и значение резидентного атрибута*

Когда значение атрибута хранится непосредственно в MFT, обращение к нему занимает значительно меньше времени. Вместо того чтобы искать файл в таблице, а затем считывать цепочку кластеров для поиска файловых данных (как, например, поступает FAT), NTFS обращается к диску только один раз и немедленно считывает данные.

Как видно из рис. 12-30, атрибуты небольшого каталога, а также небольшого файла, могут быть резидентными в MFT. Для небольшого каталога атрибут «корень индекса» содержит индекс файловых ссылок на файлы и подкаталоги этого каталога.

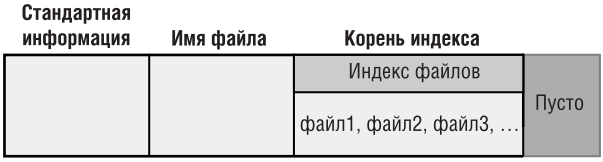


Рис. 12-30. *Запись в MFT для небольшого каталога*

Конечно, многие файлы и каталоги нельзя втиснуть в запись MFT с фиксированным размером в 1 Кб. Если некий атрибут, например файловые данные, слишком велик и не умещается в записи MFT, NTFS выделяет для него отдельные кластеры за пределами MFT. Эта область называется *группой* (run) или *экстентом* (extent). Если размер значения атрибута впоследствии расширяется (например, при добавлении в файл дополнительных данных), NTFS выделяет для новых данных еще одну группу. Атрибуты, значения которых хранятся в группах, а не в MFT, называются *нерезидентными*. Файловая система сама решает, будет атрибут резидентным или нерезидентным, и обеспечивает пользовательским процессам прозрачный доступ к этим данным.

В случае нерезидентного атрибута (им может быть атрибут данных большого файла) в его заголовке содержится информация, нужная NTFS для поиска значения атрибута на диске. На рис. 12-31 показан нерезидентный атрибут данных, хранящийся в двух группах.

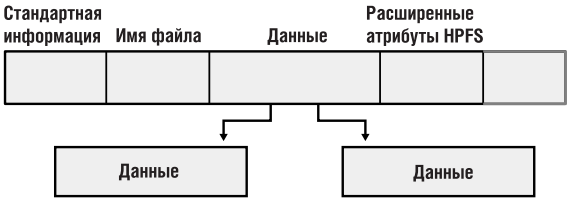


Рис. 12-31. *Запись в MFT для большого файла с двумя группами данных*

Среди стандартных атрибутов нерезидентными бывают лишь те, чей размер может увеличиваться. Для файлов такими атрибутами являются данные и список атрибутов (не показанный на рис. 12-31). Атрибуты «стандартная информация» и «имя файла» всегда резидентны.

В большом каталоге также могут быть нерезидентные атрибуты (или части атрибутов), как видно из рис. 12-32. В этом примере в записи MFT не хватает места для хранения индекса файлов, составляющих этот большой каталог. Часть индекса хранится в атрибуте корня индекса, а остальное — в нерезидентных группах, называемых *индексными буферами* (index buffers). Атрибуты корня индекса, выделенной группы индексов (index allocation) и битовой карты показаны здесь в упрощенной форме (подробнее о них см. следующий раздел). Атрибуты стандартной информации и имени файла всегда резидентны. Заголовок и по крайней мере часть значения атрибута корня индекса в случае каталогов также резидентны.

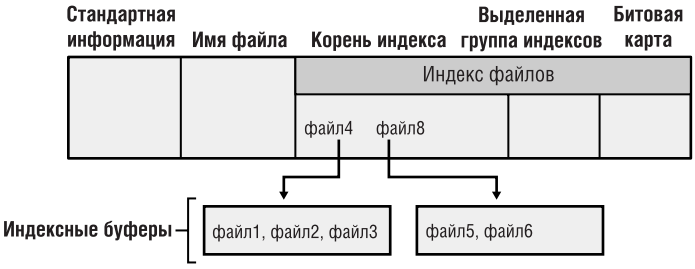


Рис. 12-32. *Запись MFT для большого каталога с нерезидентным индексом имен файлов*

Когда атрибуты файла (или каталога) не умещаются в записи MFT и для них требуется отдельное место, NTFS отслеживает выделяемые группы посредством пар сопоставлений VCN-LCN. LCN представляют последовательность кластеров на всем томе, пронумерованных от 0 до n. VCN нумеруют от 0 до m только кластеры, принадлежащие конкретному файлу. Пример нумерации кластеров в группах нерезидентного атрибута данных приведен на рис. 12-33.

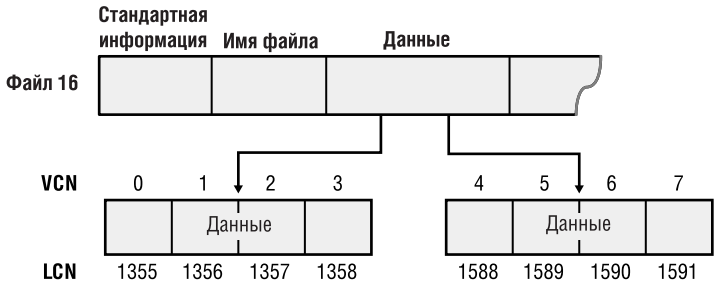


Рис. 12-33. *VCN для нерезидентного атрибута данных*

Если бы этот файл занимал больше двух групп, нумерация в третьей группе началась бы с VCN 8. Как показано на рис. 12-34, заголовок атрибута данных содержит сопоставления VCN-LCN для обоих групп, что позволяет NTFS легко находить выделенные под них области на диске.

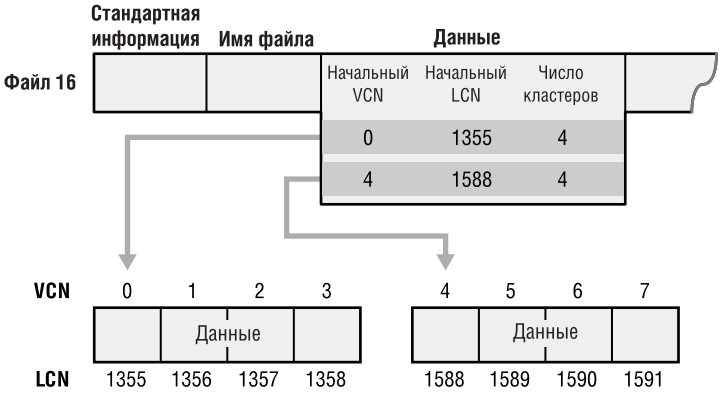


Рис. 12-34. *Сопоставления VCN-LCN для нерезидентного атрибута данных*

Хотя на рис. 12-33 показаны только группы данных, в группах могут храниться и другие атрибуты, если они не умещаются в записи MFT. Когда у файла так много атрибутов, что они не умещаются в записи MFT, для хранения дополнительных атрибутов (или заголовков в случае нерезидентных атрибутов) используется вторая запись MFT. При этом добавляется атрибут, называемый *списком атрибутов* (attribute list). Список атрибутов содержит имя и код типа каждого атрибута файла, а также файловую ссылку на запись MFT, в которой находится данный атрибут. Атрибут «список атрибутов» предназначен для тех случаев, когда файл становится настолько большим или фрагментированным, что одной записи MFT уже недостаточно для хранения большого объема сведений о сопоставлениях VCN-LCN, нужных для поиска всех групп. Список атрибутов обычно нужен файлам, у которых более 200 групп.

Сжатие данных и разреженные файлы

NTFS поддерживает сжатие по отдельным файлам, по каталогам и по томам (NTFS сжимает только пользовательские данные, не трогая метаданные файловой системы). Выяснить, сжат ли том, можно через Windows-функцию *GetVolumeInformation*. Чтобы получить реальный размер сжатого файла, используйте Windows-функцию *GetCompressedFileSize*. Наконец, проверить или изменить параметры сжатия для файла или каталога позволяет Windows-функция *DeviceIoControl* (см. управляющие коды файловой системы FSCTL_GET_COMPRESSION и FSCTL_SET_COMPRESSION в описании этой функции в Platform SDK). Учтите, что изменение степени сжатия применительно к файлу выполняется немедленно, а применительно к каталогу или тому — нет. Во втором случае степень сжатия, заданная для каталога или тома, становится степенью сжатия по умолчанию для всех новых файлов и подкаталогов, создаваемых в каталоге или на томе.

Следующий раздел является введением в сжатие данных в NTFS на примере простого случая компрессии разреженных данных. После него мы обсудим сжатие обычных и разреженных файлов.

Сжатие разреженных данных

Разреженными (sparse) называются данные (часто большого размера), в которых лишь малая часть отлична от нулевых значений. Пример разреженных данных — разреженная матрица. Как уже говорилось, для обозначения кластеров файла NTFS использует виртуальные номера кластеров (VCN) — от 0 до *m*. Каждый VCN соответствует логическому номеру кластера (LCN), который определяет местонахождение кластера на диске. На рис. 12-35 показаны группы (занимаемые участки дискового пространства) обычного (несжатого файла), а также их VCN и LCN.

Рис. 12-35. *Группы несжатого файла*

Данный файл хранится в трех группах, каждая по 4 кластера, и таким образом занимает 12 кластеров. Запись MFT для этого файла представлена на рис. 12-36. Как мы уже отмечали, для экономии места на диске атрибут данных в записи MFT содержит только одно сопоставление VCN-LCN для каждой группы, а не для каждого кластера. Тем не менее каждому VCN от 0 до 11 сопоставлен свой LCN. Первый элемент начинается с VCN 0 и охватывает 4 кластера, второй начинается с VCN 4, также охватывая 4 кластера, и т. д. Такой формат типичен для несжатого файла.

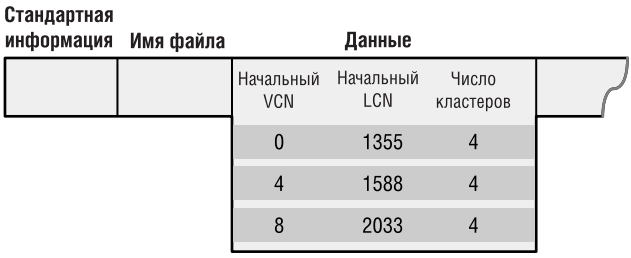


Рис. 12-36. *Запись MFT для несжатого файла*

Один из способов сжатия файла, применяемых NTFS, состоит в удалении из него длинных цепочек нулей. Если файл разрежен, он обычно сжимается до размера, составляющего лишь часть дискового пространства, необходимого для его хранения в нормальном виде. При последующей записи в этот файл NTFS выделяет пространство только для групп с ненулевыми данными.

На рис. 12-37 изображены группы сжатого разреженного файла. Обратите внимание на то, что для некоторых диапазонов VCN файла (16–31 и 64–127) дисковое пространство не выделено.

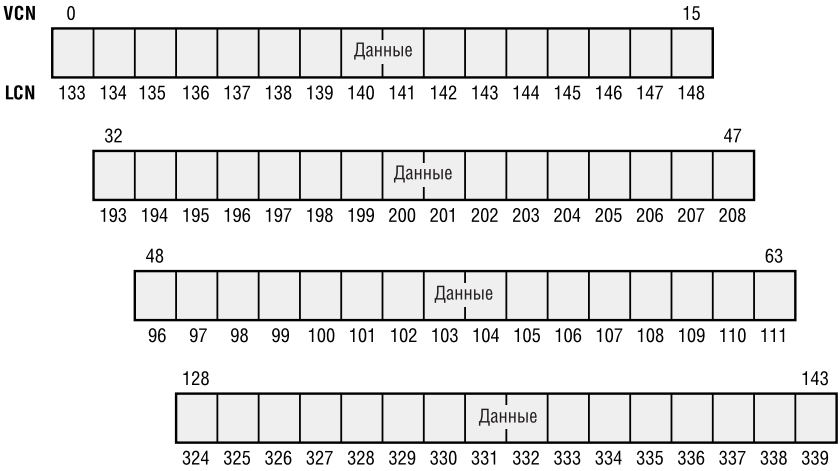


Рис. 12-37. *Группы сжатого файла, содержащего разреженные данные*

В записи MFT для этого сжатого файла пропущены блоки VCN кластеров, содержащих нули, т. е. для них не выделено дисковое пространство. Так, первый элемент данных на рис. 12-38 начинается с VCN 0 и охватывает 16 кластеров. Второй элемент перескакивает на VCN 32 и охватывает еще 16 кластеров.

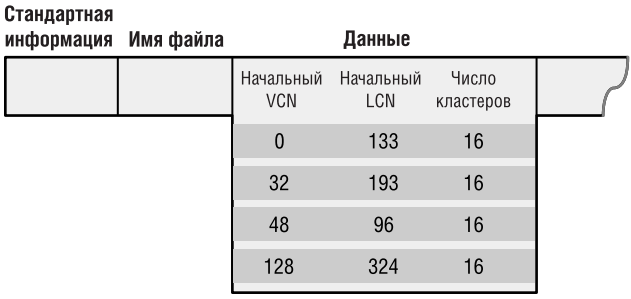


Рис. 12-38. *Запись MFT для сжатого файла, содержащего разреженные данные*

Когда программа читает данные из сжатого файла, NTFS проверяет запись MFT, чтобы выяснить, имеется ли сопоставление VCN-LCN для считываемого участка файла. Если программа обращается в невыделенную «дыру» в файле, значит, данные этой части файла состоят из нулей, и тогда NTFS возвращает нули, не обращаясь к диску. Если программа записывает в «дыру» ненулевые данные, NTFS автоматически выделяет дисковое пространство и записывает туда эти данные. Такой метод очень эффективен для разреженных файлов, содержащих много нулевых данных.

Сжатие неразреженных данных

Предыдущий пример сжатия разреженного файла довольно надуманный. Он описывает «сжатие» в том случае, когда целые области файла заполнены нулями, но на остальные файловые данные сжатие не оказывает никакого влияния. В большинстве файлов данные не являются разреженными, тем не менее их можно компрессировать по какому-нибудь алгоритму сжатия.

В NTFS пользователи могут сжимать отдельные файлы или все файлы в каталоге. (Новые файлы, создаваемые в сжатом каталоге, сжимаются автоматически. Файлы, существовавшие в каталоге до его сжатия, должны быть сжаты индивидуально.) Сжимая файл, NTFS разбивает его необработанные данные на *единицы сжатия* (compression units) длиной по 16 кластеров. Некоторые последовательности данных в файле могут сжиматься недостаточно сильно или вообще не сжиматься, поэтому для каждой единицы сжатия NTFS определяет, будет ли при ее сжатии получен выигрыш хотя бы в один кластер. Если сжатие не позволяет освободить даже один кластер, NTFS выделяет 16-кластерную группу и записывает единицу сжатия на диск, не компрессируя ее данные. Если же данные можно сжать до 15 или менее кластеров, NTFS выделяет на диске ровно столько кластеров, сколько нужно для хранения сжатых данных, после чего записывает данные на диск. Рис. 12-39 иллюстрирует сжатие файла, состоящего из четырех групп. Незакрашенные области отражают дисковое пространство, которое файл будет занимать после сжатия. Первая, вторая и четвертая группы сжимаются, а третья — нет. Но даже с одной несжатой группой достигается экономия 26 кластеров диска, т. е. длина файла уменьшается на 41%.

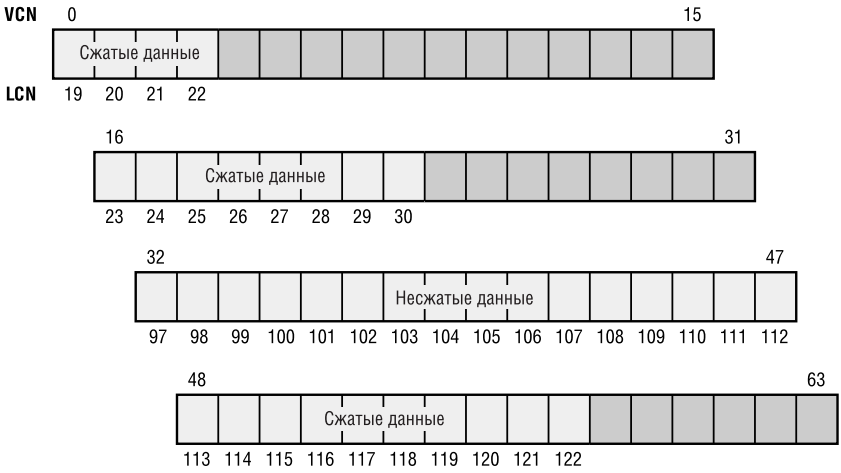


Рис. 12-39. *Группы данных сжатого файла*

ПРИМЕЧАНИЕ Хотя на схемах, приведенных в этой главе, показаны непрерывные LCN, единицы сжатия не обязательно располагаются в физически смежных кластерах. Группы, занимающие несмежные кластеры, заставляют NTFS создавать несколько более сложные записи MFT, чем показанные на рис. 12-39.

При записи данных в сжатый файл NTFS гарантирует, что каждая группа будет начинаться на виртуальной 16-кластерной границе. Таким образом, начальный VCN каждой группы кратен 16, и длина группы не превышает 16 кластеров. При работе со сжатым файлом NTFS единовременно считывает и записывает минимум одну единицу сжатия. Но при записи сжатых данных NTFS пытается помещать единицы сжатия в физически смежные области, так чтобы их можно было считывать в ходе одной операции ввода-вывода. Размер единицы сжатия в 16 кластеров выбран для уменьшения внутренней фрагментации: чем больше размер единицы сжатия, тем меньше дискового пространства нужно для хранения данных. Размер единицы сжатия, равный 16 кластерам, основан на компромиссе между минимизацией размера сжатых файлов и замедлением операций чтения для программ, использующих прямой (произвольный) доступ к содержимому файлов. При каждом промахе кэша приходится декомпрессировать эквивалент 16 кластеров (вероятность промаха кэша при прямом доступе к файлу выше, чем при последовательном). На рис. 12-40 представлена запись MFT для сжатого файла, показанного на рис. 12-39.

Одно из различий между этим сжатым файлом и сжатым разреженным файлом из более раннего примера в том, что в данном случае три группы имеют длину менее 16 кластеров. Считывание этой информации из записи MFT позволяет NTFS определить, сжаты ли данные в этом файле. Каждая группа короче 16 кластеров содержит сжатые данные, которые NTFS должна разархивировать при первом чтении группы в кэш. Группа, длина кото-

рой равна точно 16 кластерам, не содержит сжатых данных, а значит, не требует декомпрессии.

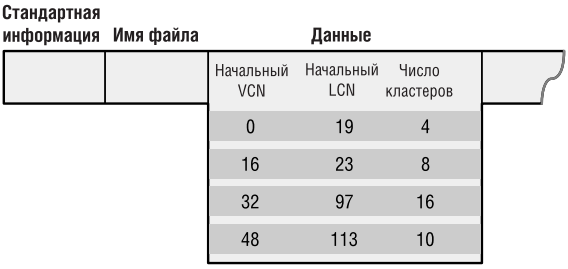


Рис. 12-40. *Запись MFT для сжатого файла*

Если группа содержит сжатые данные, NTFS разархивирует их во временный буфер и копирует в буфер вызывающей программы. Кроме того, NTFS помещает декомпрессированные данные в кэш, поэтому последующее чтение из этой группы выполняется так же быстро, как и простое чтение из кэша. Все изменения файла NTFS записывает в кэш, позволяя подсистеме отложенной записи асинхронно сжимать и записывать измененные данные на диск. Такая стратегия гарантирует, что запись в сжатый файл вызовет примерно ту же задержку, что и запись в несжатый файл.

NTFS размещает сжатый файл на диске по возможности в смежных областях. Как указывают LCN, первые две группы сжатого файла на рис. 12-38 являются физически смежными, равно как и две последних. Если две и более группы расположены последовательно, NTFS выполняет опережающее чтение с диска — как и в случае обычных файлов. Поскольку чтение и декомпрессия непрерывных файловых данных выполняются асинхронно и еще до того, как программа запросит эти данные, то в последующих операциях чтения информация извлекается непосредственно из кэша, что значительно ускоряет процесс чтения.

Разреженные файлы

Разреженные файлы (тип файлов NTFS, отличающийся от ранее описанных файлов, которые содержат разреженные данные) по сути являются сжатыми файлами, неразреженные данные которых NTFS не сжимает. Однако NTFS обрабатывает данные группы из записи MFT разреженного файла так же, как и в случае сжатых файлов, состоящих из разреженных и неразреженных данных.

Файл журнала изменений

Файл журнала изменений, \$Extend\$UsnJrnl, является разреженным файлом, который NTFS создает, только когда приложение активизирует регистрацию изменений. Журнал хранит записи изменений в потоке данных $J. Эти записи включают следующую информацию об изменениях файлов или каталогов:

- время изменения;
- тип изменения (удаление, переименование, увеличение размера и т. д.);

- атрибуты файла или каталога;
- имя файла или каталога;
- номер файловой ссылки файла или каталога;
- номер файловой ссылки родительского каталога файла.

Поскольку журнал является разреженным, он никогда не переполняется; когда его размер на диске достигает определенного максимума, NTFS начинает просто обнулять файловые данные, предшествующие текущему блоку информации об изменениях, как показано на рис. 12-41. Но, чтобы предотвратить постоянное изменение размера журнала, NTFS уменьшает его, только когда он в два раза превосходит установленный максимум.

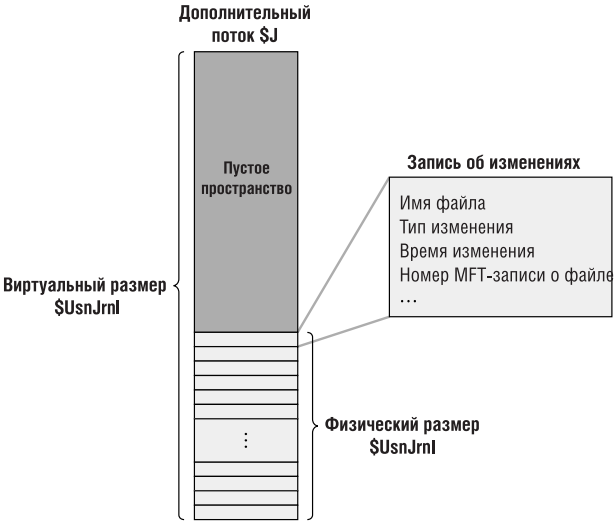


Рис. 12-41. *Выделение пространства для журнала изменений ($UsnJrnl)*

Индексация

В NTFS каталог — это просто индекс имен файлов, т. е. совокупность имен файлов (с соответствующими файловыми ссылками), организованная таким образом, чтобы обеспечить быстрый доступ. Для создания каталога NTFS индексирует атрибуты «имя файла» из этого каталога. Запись MFT для корневого каталога тома показана на рис. 12-42.

С концептуальной точки зрения, элемент MFT для каталога содержит в своем атрибуте «корень индекса» отсортированный список файлов каталога. Но для больших каталогов имена файлов на самом деле хранятся в индексных буферах размером по 4 Кб, которые содержат и структурируют имена файлов. Индексные буферы реализуют структуру данных типа «b+ tree», которая позволяет свести к минимуму число обращений к диску при поиске какого-либо файла, особенно в больших каталогах. Атрибут «корень индекса» содержит первый уровень структуры b+ tree (подкаталоги корневого каталога) и

указывает на индексные буферы, содержащие следующий уровень (другие подкаталоги или файлы).

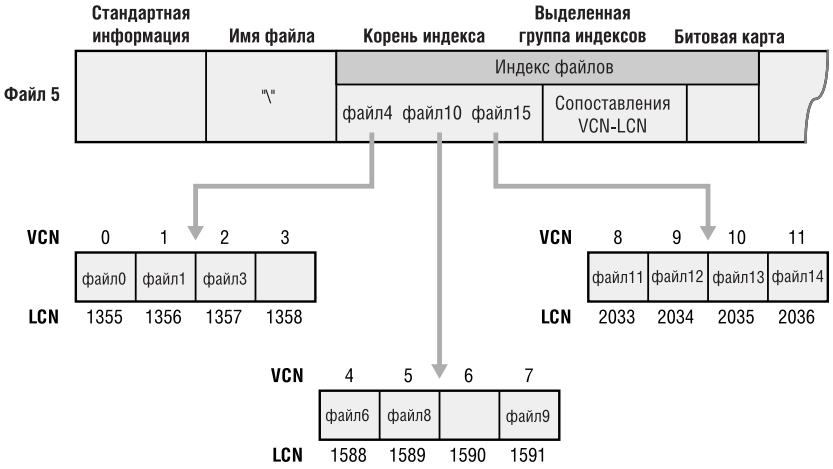


Рис. 12-42. *Индекс имен файлов для корневого каталога тома*

На рис. 12-42 в атрибуте «корень индекса» и индексных буферах показаны только имена файлов (например, *файлб*), но каждая запись индекса содержит и файловую ссылку на запись MFT, описывающую данный файл, плюс метку времени и информацию о размере файла. NTFS дублирует метку времени и информацию о размере файла из записи MFT для файла. Такой подход, используемый файловыми системами FAT и NTFS, требует записи обновленной информации в два места. Но даже при этом просмотр каталогов существенно ускоряется, поскольку файловая система может сообщать метки времени и размеры файлов, не открывая каждый файл в каталоге.

Атрибут «выделенная группа индексов» сопоставляет VCN групп индексных буферов с LCN, которые указывают, в каком месте диска находятся индексные буферы, а битовая карта используется для учета того, какие VCN в индексных буферах заняты, а какие свободны. На рис. 12-42 на каждый VCN, т. е. на каждый кластер, приходится по одной записи для файла, но на самом деле кластер содержит несколько записей. Каждый индексный буфер размером 4 Кб может содержать 20–30 записей для имен файлов.

Структура данных b+ tree — это разновидность сбалансированного дерева, идеальная для организации отсортированных данных, хранящихся на диске, так как позволяет минимизировать количество обращений к диску при поиске заданного элемента. В MFT атрибут корня индекса для каталога содержит несколько имен файлов, выступающих в качестве индексов для второго уровня b+ tree. С каждым именем файла в атрибуте корня индекса связан необязательный указатель индексного буфера. Этот индексный буфер содержит имена файлов, которые с точки зрения лексикографии меньше данного имени. Например, на рис. 12-42 *файл4* — это элемент первого уровня b+ tree. Он указывает на индексный буфер, содержащий имена файлов,

которые лексикографически меньше имени в этом элементе, — *файл0*, *файл1* и *файл3*. Обратите внимание, что использованные в этом примере имена (*файл1*, *файл2* и др.) не являются буквальными, — они просто иллюстрируют относительное размещение файлов, лексикографически упорядоченных в соответствии с показанной последовательностью.

Хранение имен файлов в структурах вида b+ tree дает несколько преимуществ. Поиск в каталоге выполняется быстрее, так как имена файлов хранятся в отсортированном порядке. А когда высокоуровневое программное обеспечение перечисляет файлы в каталоге, NTFS возвращает уже отсортированные имена. Наконец, поскольку b+ tree имеет тенденцию к росту в ширину, а не в глубину, скорость поиска не уменьшается с увеличением размера каталога.

Кроме индексации имен, NTFS обеспечивает универсальную индексацию данных, и некоторая функциональность NTFS (в том числе идентификации объектов, отслеживания квот и консолидированной защиты) использует индексацию для управления внутренними данными.

Идентификаторы объектов

Кроме идентификатора объекта, назначенного файлу или каталогу и хранящегося в атрибуте $OBJECT_ID записи MFT, NTFS также запоминает соответствие между идентификаторами объектов и номерами их файловых ссылок в индексе $O файла метаданных \$Extend\$ObjId. Элементы индекса сортируются по значениям идентификатора объекта, благодаря чему NTFS может быстро находить файл по его идентификатору. Таким образом, используя недокументированную функциональность, приложения могут открывать файл или каталог по идентификатору объекта. На рис. 12-43 показана взаимосвязь между файлом метаданных $Objid и атрибутами $OBJECT_ID в MFT-записях.

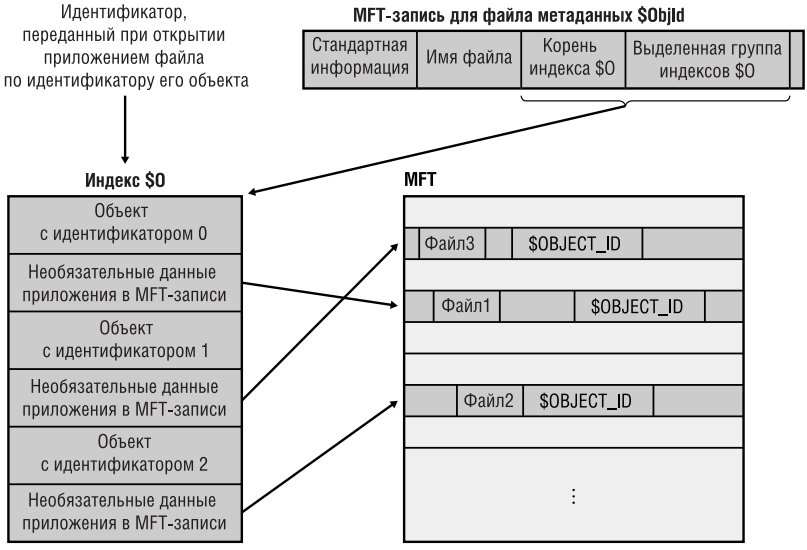


Рис. 12-43. *Взаимосвязь $ObjId и $OBJECT_ID*

Отслеживание квот

NTFS хранит информацию о квотах в файле метаданных \$Extend\$Quota, который состоит из индексов $O и $Q. Структура этих индексов показана на рис. 12-44. NTFS не только присваивает каждому дескриптору защиты уникальный внутренний идентификатор защиты, но и назначает каждому пользователю уникальный идентификатор. Когда администратор задает квоты для пользователя, NTFS создает идентификатор этого пользователя, соответствующий его SID. NTFS создает в индексе $O запись, сопоставляющую SID с идентификатором пользователя, и сортирует этот индекс по идентификаторам пользователей; в индексе $Q создается запись, управляющая квотами (quota control entry). Эта запись содержит лимиты, выделенные пользователю, а также объем дискового пространства, отведенный ему на данном томе.

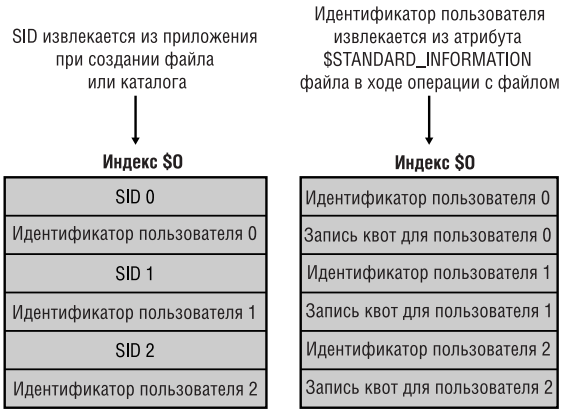


Рис. 12-44. *Индексация $Quota*

Когда приложение создает файл или каталог, NTFS получает SID пользователя этого приложения и ищет соответствующий идентификатор пользователя в индексе $O. Этот идентификатор записывается в атрибут $STANDARD_INFORMATION нового файла или каталога. Затем NTFS просматривает запись квот в индексе $Q и определяет, не превышает ли выделенное дисковое пространство установленные для данного пользователя лимиты. Когда новое дисковое пространство, выделяемое пользователю, превышает пороговое значение, NTFS предпринимает соответствующие меры, например, записывает событие в журнал System (Система) или отклоняет запрос на создание файла или каталога.

Консолидированная защита

NTFS всегда поддерживала средства защиты, которые позволяют администратору указывать, какие пользователи могут обращаться к определенным файлам и каталогам, а какие — не могут. В версиях NTFS до Windows 2000 каждый файл и каталог хранит дескриптор защиты в своем атрибуте защиты. Но в большинстве случаев администратор применяет одинаковые пара-

метры защиты к целому дереву каталогов, что приводит к дублированию дескрипторов защиты во всех файлах и подкаталогах этого дерева каталогов. В многопользовательских средах, например в Windows 2000 Server со службой Terminal Services, такое дублирование может потребовать слишком большого пространства на диске, поскольку дескрипторы защиты будут содержать элементы для множества учетных записей. NTFS в Windows 2000 и более поздних версиях ОС оптимизируют использование дискового пространства дескрипторами защиты за счет применения централизованного файла метаданных $Secure, в котором хранится только один экземпляр каждого дескриптора защиты на данном томе.

Файл $Secure содержит два атрибута индексов ($SDH и $SII), а также атрибут потока данных $SDS, как показано на рис. 12-45. NTFS назначает каждому уникальному дескриптору защиты на томе внутренний для NTFS идентификатор защиты (не путать с SID, который уникально идентифицирует учетные записи компьютеров и пользователей) и хэширует дескриптор защиты по простому алгоритму. Хэш является потенциально неуникальным «стенографическим» представлением дескриптора. Элементы в индексе $SDH увязывают эти хэши с местонахождением дескриптора защиты внутри атрибута данных $SDS, а элементы индекса $SII сопоставляют NTFS-идентификаторы защиты с местонахождением дескриптора защиты в атрибуте данных $SDS.

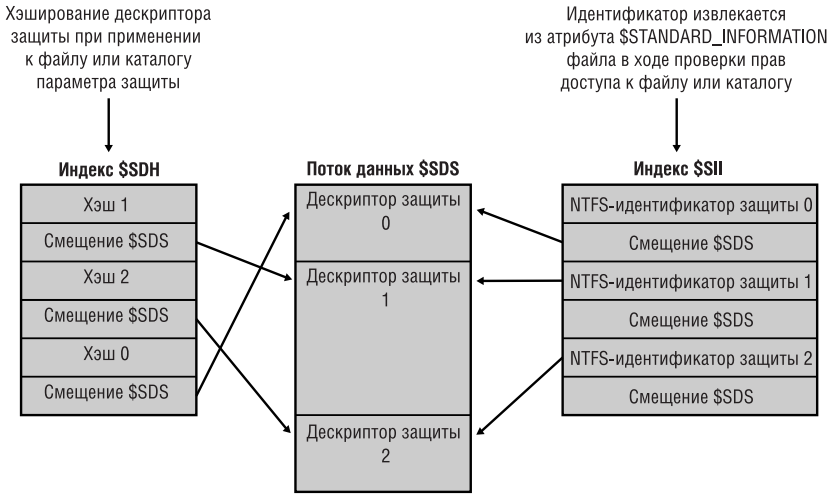


Рис. 12-45. *Индексация $Secure*

Когда вы применяете дескриптор защиты к файлу или каталогу, NTFS получает хэш этого дескриптора и просматривает индекс $SDH, пытаясь найти совпадение. NTFS сортирует элементы индекса $SDH по хэшам дескрипторов защиты и хранит эти элементы в структуре вида b+ tree. Обнаружив совпадение для дескриптора в индексе $SDH, NTFS находит смещение дескриптора защиты от смещения элемента и считывает дескриптор из атрибута $SDS. Если хэши совпадают, а дескрипторы — нет, NTFS ищет следующее совпадение в

индексе $SDH. Когда NTFS находит точное совпадение, файл или каталог, к которому вы применяете дескриптор защиты, может ссылаться на существующий дескриптор в атрибуте $SDS. Тогда NTFS считывает NTFS-идентификатор защиты из элемента $SDH и сохраняет его в атрибуте $STANDARD_ INFORMATION файла или каталога. Атрибут $STANDARD_INFORMATION, имеющийся у всех файлов и каталогов, хранит базовую информацию о файле, в том числе его атрибуты, временные метки и идентификатор защиты.

Если NTFS не обнаруживает в индексе $SDH элемент с дескриптором защиты, совпадающим с тем, который вы применяете, значит, ваш дескриптор уникален в пределах данного тома, и NTFS присваивает ему новый внутренний идентификатор защиты. Такие идентификаторы являются 32-битными значениями, но SID обычно в несколько раз больше, поэтому представление SID в виде NTFS-идентификаторов защиты экономит место в атрибуте $STANDARD_INFORMATION. NTFS включает дескриптор защиты в атрибут $SDS, который сортируется в структуру вида b+ tree по NTFS-идентификатору защиты, а потом добавляет его в элементы индексов $SDH и $SII, ссылающиеся на смещение дескриптора в данных $SDS.

Когда приложение пытается открыть файл или каталог, NTFS использует индекс $SII для поиска дескриптора защиты этого файла или каталога. NTFS считывает внутренний идентификатор защиты файла или каталога из атрибута $STANDARD_INFORMATION записи MFT. Затем по индексу $SII файла $Secure она находит элемент с нужным идентификатором в атрибуте $SDS. По смещению в атрибуте $SDS NTFS считывает дескриптор защиты и завершает проверку прав доступа. NTFS хранит последние 32 дескриптора защиты, к которым было обращение, вместе с их элементами $SII в кэше, чтобы впоследствии была возможность обращаться только к файлу $Secure.

NTFS не удаляет элементы в файле $Secure, даже если на них не ссылаются никакие файлы или каталоги на томе. Это приводит лишь к незначительному увеличению места, занимаемого файлом $Secure, так как на большинстве томов, даже если они используются уже весьма долго, уникальных дескрипторов защиты сравнительно немного.

Механизм универсальной индексации позволяет NTFS повторно использовать дескрипторы защиты для файлов и каталогов с одинаковыми параметрами защиты. По индексу $SII NTFS быстро находит дескриптор защиты в файле $Secure при проверках прав доступа, а по индексу $SDH определяет, хранится ли применяемый к файлу или каталогу дескриптор защиты в файле $Secure, и, если да, использует этот дескриптор повторно.

Точки повторного разбора

Как уже упоминалось, *точка повторного разбора* представляет собой блок данных повторного разбора, определяемых приложениями; длина такого блока может быть до 16 Кб. В нем содержится 32-битный тэг повторного разбора, который хранится в атрибуте $REPARSE_POINT файла или каталога. Всякий раз, когда приложение создает или удаляет точку повторного разбора, NTFS обновляет файл метаданных \$Extend\$Reparse, в котором она

хранит элементы, идентифицирующие номера записей о файлах и каталогах с точками повторного разбора. Централизованное хранение записей позволяет NTFS предоставлять приложениям интерфейсы для перечисления либо всех точек повторного разбора на томе, либо только точек заданного типа, например точек монтирования (подробнее о точках монтирования см. главу 10). Файл \$Extend\$Reparse использует NTFS-механизм универсальной индексации, сортируя элементы файлов (в индексе с именем $R) по тэгам повторного разбора.

Поддержка восстановления в NTFS

Поддержка восстановления в NTFS гарантирует, что в случае отказа электропитания или аварии системы ни одна операция файловой системы (транзакция) не останется незавершенной; при этом структура дискового тома будет сохранена. NTFS включает утилиту Chkdsk, которая позволяет устранять последствия катастрофических повреждений диска, вызванных аппаратными ошибками ввода-вывода (например, из-за аварийных секторов на диске, электрических аномалий или сбоев в работе диска) либо ошибками в программном обеспечении. Наличие средств восстановления NTFS уменьшает потребность в использовании Chkdsk.

Как уже упоминалось в разделе «Восстанавливаемость», NTFS использует схему на основе обработки транзакций. Эта стратегия гарантирует полное восстановление диска, которое производится исключительно быстро (за считанные секунды) даже в случае самых больших дисков. NTFS ограничивается восстановлением данных файловой системы, гарантируя, что пользователь по крайней мере никогда не потеряет весь том из-за повреждения файловой системы. Но, если приложение не выполнило определенных действий, например не сбросило на диск кэшированные файлы, NTFS не гарантирует полное восстановление пользовательских данных в случае краха. Защита пользовательских данных на основе технологии обработки транзакций предусматривается в большинстве СУБД для Windows, например в Microsoft SQL Server. Microsoft не стала реализовать восстановление пользовательских данных на уровне файловой системы, так как приложения обычно поддерживают свои схемы восстановления, оптимизированные под тот тип данных, с которыми они работают.

В следующих разделах рассказывается об эволюции средств обеспечения надежности файловой системы, и в этом контексте вводится концепция восстанавливаемых файловых систем с подробным обсуждением схемы протоколирования транзакций, за счет которой NTFS регистрирует изменения в структурах данных файловой системы. Кроме того, мы поясним, как NTFS восстанавливает том в случае сбоя системы.

Эволюция архитектуры файловых систем

Создание восстанавливаемой файловой системы можно рассматривать как еще один шаг в эволюции архитектуры файловых систем. В прошлом были

распространены два основных подхода к организации поддержки ввода-вывода и кэширования в файловых системах: *точная запись* (careful write) и *отложенная* (lazy write). В файловых системах, которые разрабатывались для VAX/VMS (права на нее перешли от DEC к Compaq) и некоторых других закрытых операционных систем, использовался алгоритм точной записи, тогда как в файловой системе HPFS операционной системы OS/2 и большинстве файловых систем UNIX применялась схема отложенной записи.

Далее мы кратко рассмотрим два типа файловых систем, наиболее распространенные в настоящее время, и присущие им внутренние противоречия между безопасностью и производительностью. В третьем подразделе будет описан подход к восстановлению, принятый в NTFS, и его отличие от двух вышеупомянутых стратегий.

Файловые системы с точной записью

В случае аварии операционной системы или сбоя электропитания операции ввода-вывода, выполняемые в данный момент, немедленно прерываются. В зависимости от того, какие операции ввода-вывода выполнялись и насколько далеко продвинулось их выполнение, такое прерывание может нарушить целостность файловой системы. В данном контексте нарушение целостности — это повреждение файловой системы: например, имя файла появляется в списке каталога, но файловая система не знает, где он находится, или не может обратиться к его содержимому. В самых тяжелых случаях повреждение файловой системы может привести к потере всего тома.

Файловая система с точной записью не пытается предотвратить нарушения целостности. Вместо этого она упорядочивает операции записи так, что авария системы в худшем случае вызовет предсказуемые, некритичные рассогласования, которые файловая система сможет в любой момент устранить.

Когда файловая система (какого бы типа она ни была) получает запрос на обновление содержимого диска, она должна выполнить несколько подопераций, прежде чем обновление будет завершено. В файловой системе, использующей стратегию точной записи, эти подоперации всегда записывают свои данные на диск последовательно. При выделении дискового пространства, например для файла, файловая система сначала устанавливает некоторые биты в своей битовой карте, после чего выделяет место для файла. Если сбой питания происходит сразу после того, как были установлены эти биты, файловая система точной записи теряет доступ к той части диска, которая была представлена установленными битами, но существующие данные не разрушаются.

Упорядочение операций записи также означает, что запросы на ввод-вывод выполняются в порядке их поступления. Если один процесс выделяет дисковое пространство и вскоре после этого другой процесс создает файл, файловая система с точной записью завершает выделение дискового пространства до того, как начнет создавать файл, — иначе перекрытие подопераций из двух запросов ввода-вывода могло бы привести к нарушению целостности.

ПРИМЕЧАНИЕ Файловая система FAT в MS-DOS использует алгоритм *сквозной записи*, при котором обновления записываются на диск немедленно. В отличие от точной записи этот метод не требует от файловой системы упорядочения операций вывода для предотвращения нарушения целостности.

Основное преимущество файловых систем с точной записью в том, что в случае сбоя дисковый том остается целостным и его можно использовать дальше — немедленный запуск утилиты исправления тома не обязателен. Такая утилита нужна для исправления предсказуемых, неразрушительных нарушений целостности диска, которые возникли в результате сбоя, но ее можно запустить в удобное время, обычно при перезагрузке системы.

Файловые системы с отложенной записью

Файловая система с точной записью жертвует производительностью ради надежности. Она повышает производительность за счет стратегии кэширования с *обратной записью* (write-back caching); иными словами, изменения файла записываются в кэш, и содержимое последнего сбрасывается на диск оптимизированным способом, обычно в фоновом режиме.

Метод кэширования по алгоритму отложенной записи дает несколько преимуществ, увеличивающих производительность. Во-первых, уменьшается общее число операций записи на диск. Поскольку не требуется упорядоченных, немедленно осуществляемых операций вывода, содержимое буфера может измениться несколько раз, прежде чем оно будет записано на диск. Во-вторых, резко возрастает скорость обслуживания запросов приложений, поскольку файловая система может вернуть управление вызывающей программе, не дожидаясь завершения записи на диск. Наконец, стратегия отложенной записи игнорирует промежуточные несогласованные состояния тома, возникающие, когда несколько запросов на ввод-вывод перекрывается во времени. Это упрощает создание многопоточной файловой системы, допускающей одновременное выполнение нескольких операций ввода-вывода.

Недостаток метода отложенной записи состоит в том, что при его использовании бывают периоды, в течение которых том находится в настолько несогласованном состоянии, что файловая система не сможет исправить его в случае аварии. Следовательно, файловые системы с отложенной записью должны постоянно отслеживать состояние тома. В общем случае отложенная запись дает выигрыш в производительности по сравнению с точной записью — за счет большего риска и неудобств для пользователя при сбое системы.

Восстанавливаемые файловые системы

Восстанавливаемая файловая система типа NTFS превосходит по надежности файловые системы с точной записью и при этом достигает уровня производительности файловых систем с отложенной записью. Восстанавливаемая файловая система гарантирует сохранение целостности тома; с этой целью используется журнал изменений, изначально созданный для обработ-

ки транзакций. В случае аварии операционной системы такая файловая система восстанавливает целостность, выполняя процедуру восстановления на основе информации из файла журнала. Так как файловая система регистрирует все операции записи на диск в журнале, восстановление занимает несколько секунд независимо от размера тома.

Процедура восстановления в восстанавливаемой файловой системе является точной и гарантирует возвращение тома в согласованное состояние. Неадекватные результаты восстановления, характерные для файловых систем с отложенной записью, в NTFS исключены.

За высокую надежность восстанавливаемой файловой системы приходится расплачиваться. При каждой транзакции, изменяющей структуру тома, в файл журнала требуется заносить по одной записи на каждую подоперацию транзакции. Файловая система уменьшает издержки протоколирования за счет объединения записей файла журнала в пакеты: за одну операцию ввода-вывода в журнал добавляется сразу несколько записей. Кроме того, восстанавливаемая файловая система может применять алгоритмы оптимизации, используемые файловыми системами с отложенной записью. Она может даже увеличить интервалы между сбросами кэша на диск, так как файловую систему можно восстановить, если авария произошла до того, как изменения переписаны из кэша на диск. Такой рост в производительности кэша компенсирует и часто даже перевешивает издержки, вызванные протоколированием транзакций.

Ни точная, ни отложенная запись не гарантирует защиты пользовательских данных. Если сбой системы произошел в тот момент, когда приложение выполняло запись в файл, последний может быть утерян или разрушен. Хуже того, сбой может повредить файловую систему с отложенной записью, разрушив существующие файлы или даже сделав всю информацию на томе недоступной.

Восстанавливаемая файловая система NTFS использует стратегию, повышающую ее надежность по сравнению с традиционными файловыми системами. Во-первых, восстанавливаемость NTFS гарантирует, что структура тома не будет разрушена, так что в случае сбоя системы все файлы останутся доступными.

Во-вторых, хотя NTFS не гарантирует сохранности пользовательских данных в случае сбоя системы (некоторые изменения в кэше могут быть потеряны), приложения могут использовать преимущества сквозной записи и сброса кэша NTFS для гарантии того, что изменения файлов будут записываться на диск в должное время. Как сквозная запись (принудительная немедленная запись на диск), так и сброс кэша (принудительная запись на диск содержимого кэша) — операции вполне эффективные. NTFS не требуется дополнительного ввода-вывода для сброса на диск изменений нескольких различных структур данных файловой системы, так как изменения в этих структурах регистрируются в файле журнала (в ходе единственной операции записи); если произошел сбой и содержимое кэша потеряно, изменения файловой системы могут быть восстановлены по информации из журнала. Более того, NTFS в

отличие от FAT гарантирует, что сразу после операции сквозной записи или сброса кэша пользовательские данные останутся целостными и будут доступны, даже если вслед за этим произойдет сбой системы.

Протоколирование

Восстанавливаемость NTFS обеспечивается методикой обработки транзакций, называемой *протоколированием* (logging). Прежде чем выполнить над содержимым диска подоперации какой-либо транзакции, изменяющей важные структуры данных файловой системы, NTFS регистрирует ее в файле журнала. Таким образом, в случае сбоя системы незавершенные транзакции можно повторить или отменить после перезагрузки компьютера. В технологии обработки транзакций эта методика называется *опережающим протоколированием* (write-ahead logging). В NTFS транзакции, к которым относятся, в частности, запись на диск или удаление файла, могут состоять из нескольких подопераций.

Сервис файла журнала

Сервис файла журнала (log file service, LFS) — это набор процедур режима ядра, локализованных в драйвере NTFS, который она использует для доступа к файлу журнала. Хотя LFS изначально был разработан для того, чтобы предоставлять средства протоколирования и восстановления более чем одному клиенту, он используется только NTFS. Вызывающая программа, в данном случае NTFS, передает LFS указатель на открытый объект «файл», который определяет файл, выступающий в роли журнала. LFS либо инициализирует новый журнал, либо вызывает диспетчер кэша для доступа к существующему журналу через кэш, как показано на рис. 12-46.

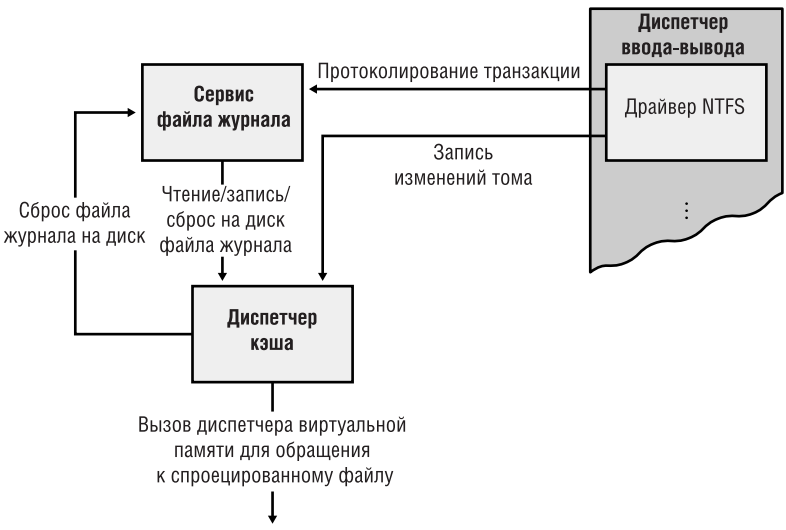


Рис. 12-46. *Сервис файла журнала (LFS)*

LFS делит файл журнала на две части: *область перезапуска* (restart area) и «безразмерную» *область протоколирования* (logging area) (рис. 12-47).

Рис. 12-47. *Области файла журнала*

NTFS вызывает LFS для чтения и записи области перезапуска. В этой области NTFS хранит информацию о контексте, например позицию в области протоколирования, откуда начнется чтение при восстановлении после сбоя системы. На тот случай, если область перезапуска будет разрушена или по каким-либо причинам станет недоступной, LFS создает ее копию. Остальная часть журнала транзакций — это область протоколирования, в которой находятся записи транзакций, обеспечивающие восстановление тома после сбоя. LFS создает иллюзию бесконечности журнала транзакций за счет его циклического повторного использования (в то же время не перезаписывая нужную информацию). Для идентификации записей, помещенных в журнал, LFS использует *номера логической последовательности* (logical sequence number, LSN). Циклически используя журнал, LFS увеличивает значения LSN. NTFS представляет LSN в виде 64-битных чисел, поэтому число возможных LSN настолько велико, что практически может считаться бесконечным.

NTFS никогда не выполняет чтение/запись транзакций в журнал напрямую. LFS предоставляет сервисы, которые NTFS вызывает для открытия файла журнала, помещения в него записей, чтения записей из журнала в прямом и обратном порядке, сброса записей журнала до заданного LSN или установки логического начала журнала на больший LSN. В процессе восстановления NTFS вызывает LFS для чтения записей журнала в прямом направлении, чтобы повторить все транзакции, которые запротоколированы в журнале, но не записаны на диск в момент сбоя. NTFS также обращается к LFS для чтения записей в обратном направлении, чтобы отменить (или откатить) все транзакции, не полностью запротоколированные перед аварией системы, и установить начало файла журнала на запись с бóльшим LSN после того, как старые записи журнала стали не нужны.

Вот как система обеспечивает восстановление тома.

1. Сначала NTFS вызывает LFS для записи в кэшируемый файл журнала любых транзакций, модифицирующих структуру тома.
2. NTFS модифицирует том (также в кэше).
3. Диспетчер кэша сообщает LFS сбросить файл журнала на диск. (Этот сброс реализуется LFS путем обратного вызова диспетчера кэша с указанием страниц памяти, подлежащих выводу на диск; см. рис. 12-46.)

4. Сбросив на диск журнал транзакций, диспетчер кэша записывает на диск изменения тома (результаты операций над метаданными).

Эта последовательность действий гарантирует: если завершить изменение файловой системы не удастся, соответствующие транзакции можно будет считать из журнала и либо повторить, либо отменить в процессе восстановления файловой системы.

Восстановление файловой системы начинается автоматически при первом обращении к дисковому тому после перезагрузки. NTFS проверяет, применялись ли к тому транзакции, запротоколированные в журнале до сбоя, и, если нет, повторяет их. NTFS гарантирует и отмену транзакций, не полностью запротоколированных до сбоя, так что вызываемые ими изменения не появятся на томе.

Типы записей журнала

LFS позволяет своим клиентам помещать в журналы транзакций записи любого типа. NTFS использует несколько типов записей, два из которых — *записи модификации* (update records) и *записи контрольной точки* (checkpoint records) — будут рассмотрены в этом разделе.

Записи модификации К ним относится большинство записей, которые NTFS помещает в журнал транзакций. Каждая такая запись содержит два вида информации.

- **Информация для повтора (redo information)** Описывает, как вновь применить к дисковому тому одну подоперацию полностью запротоколированной транзакции, если сбой системы произошел до того, как транзакция была переписана из кэша на диск.
- **Информация для отмены (undo information)** Описывает, как обратить изменения, вызванные одной подоперацией транзакции, которая в момент сбоя была запротоколирована лишь частично.

На рис. 12-48 показаны три записи модификации в файле журнала. Каждая запись представляет одну подоперацию транзакции, создающей новый файл. Информация для повтора в каждой записи модификации сообщает NTFS, как повторно применить данную подоперацию к дисковому тому, а информация для отмены — как откатить (отменить) эту подоперацию.

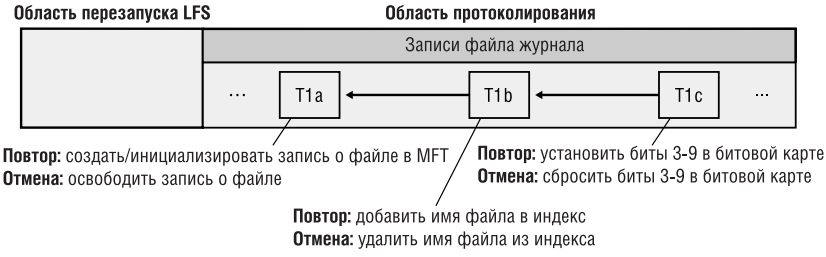


Рис. 12-48. *Записи модификации в файле журнала*

После протоколирования транзакции (в данном примере — вызовом LFS до помещения трех записей модификации в файл журнала) NTFS выполняет в кэше подоперации этой транзакции, изменяющие том. Закончив обновление кэша, NTFS помещает в журнал еще одну запись, которая помечает всю транзакцию как завершенную. Эта подоперация известна как *фиксация транзакции* (committing transaction). После фиксации транзакции NTFS гарантирует, что все вызванные ею модификации будут отражены на томе, даже если после фиксации произойдет сбой операционной системы.

При восстановлении после сбоя системы NTFS просматривает журнал и повторяет все зафиксированные транзакции. Даже если NTFS и завершила транзакцию до момента сбоя системы, ей неизвестно, были ли изменения тома своевременно переписаны на диск диспетчером кэша. Модификации, выполненные в кэше, могли быть потеряны при сбое. Следовательно, NTFS выполняет зафиксированную транзакцию снова, чтобы гарантировать актуальность состояния диска.

После повтора всех зафиксированных транзакций NTFS отыскивает в журнале такие, которые не были зафиксированы к моменту сбоя, и откатывает каждую запротоколированную подоперацию. В случае, представленном на рис. 12-48, NTFS вначале должна была бы отменить подоперацию T1*c*, после чего перейти по указателям назад и отменить T1*b*. Переход по указателям в обратном направлении и отмена подопераций продолжались бы до тех пор, пока NTFS не достигла бы первой подоперации транзакции. Следуя указателям, NTFS определяет, сколько и какие записи модификации нужно отменить для того, чтобы откатить транзакцию.

Информация для повтора и отмены может быть выражена либо физически, либо логически. Физическое описание задает модификации тома как диапазоны байтов на диске, которые следует изменить, переместить и т. д., а логическое — представляет модификации в терминах операций, например «удалить файл A.dat». Как самый низкий уровень программного обеспечения, поддерживающего структуру файловой системы, NTFS использует записи модификации с физическими описаниями. Однако для программного обеспечения обработки транзакций и других приложений записи модификации в логическом виде могут быть удобнее, поскольку логическое представление обновлений тома компактнее физических. Логическое описание требует участия NTFS в выполнении действий, связанных с такими операциями, как удаление файла.

NTFS генерирует записи модификации (обычно несколько) для каждой из следующих транзакций:

- создание файла;
- удаление файла;
- расширение файла;
- урезание файла;
- установка файловой информации;

- переименование файла;
- изменение прав доступа к файлу.

Информация для повтора и отмены в записи модификации должна быть очень точной, иначе при отмене транзакции, восстановлении после сбоя системы или даже в ходе нормальной работы NTFS может попытаться повторить транзакцию, которая уже выполнена, или, наоборот, отменить транзакцию, которая никогда не выполнялась либо уже отменена. Аналогичным образом NTFS может попытаться повторить или отменить транзакцию, включающую нескольких записей модификации, которые не все были применены к диску. Формат записей модификации должен гарантировать, что лишние операции повтора или отмены будут *идемпотентными* (idempotent), т. е. дадут нейтральный эффект. Например, установка уже установленного бита, не оказывает никакого действия, но изменение на противоположное значения бита, которое уже изменено, — оказывает. Файловая система также должна корректно обрабатывать переходные состояния тома.

Записи контрольной точки Помимо записей модификации NTFS периодически помещает в файл журнала запись контрольной точки, как показано на рис. 12-49.

Рис. 12-49. *Запись контрольной точки в файле журнала*

Запись контрольной точки помогает NTFS определить, какая обработка нужна для восстановления тома, если сбой произошел сразу после добавления этой записи в журнал. Благодаря записи контрольной точки NTFS, например, знает, как далеко назад ей нужно пройти по журналу, чтобы начать восстановление. Добавив новую запись контрольной точки, NTFS записывает ее LSN в область перезапуска, так что, начиная восстановление после сбоя системы, она быстро находит самую последнюю запись контрольной точки.

Хотя LFS представляет NTFS, будто журнал транзакций безразмерен, на самом деле он не бесконечен. Значительный размер журнала транзакций и частая вставка записей контрольной точки (операция, обычно освобождающая место в файле журнала) делают вероятность его переполнения достаточно малой. Тем не менее LFS учитывает такую возможность, отслеживая несколько значений:

- размер свободного пространства в журнале;
- размер пространства, необходимого для добавления в журнал следующей записи и для отмены этого действия (если это вдруг потребуется);

- размер пространства, нужного для отката всех активных (не зафиксированных) транзакций (если это вдруг потребуется).

Если в журнале не хватает места для суммы последних двух значений из списка, LFS сообщает об ошибке переполнения файла журнала, и NTFS генерирует исключение. Обработчик исключений NTFS откатывает текущую транзакцию и помещает ее в очередь для последующего перезапуска.

Чтобы освободить пространство в журнале транзакций, NTFS должна временно приостановить ввод-вывод в системе. Для этого она блокирует создание и удаление файлов, после чего запрашивает монопольный доступ ко всем системным файлам и разделяемый доступ ко всем пользовательским файлам. Постепенно активные транзакции либо завершаются успешно, либо вызывают исключение переполнения файла журнала. В последнем случае NTFS откатывает их, а затем помещает в очередь.

Заблокировав вышеописанным способом выполнение транзакций при операциях над файлами, NTFS вызывает диспетчер кэша для сброса на диск еще не записанных туда данных, в том числе данных файла журнала. После того как все успешно записано на диск, данные в журнале NTFS становятся ненужными. NTFS устанавливает начало журнала на текущую позицию, что делает журнал «пустым». Затем NTFS перезапускает транзакции, поставленные ранее в очередь. Так что за исключением короткой паузы в обработке ввода-вывода ошибка переполнения файла журнала не оказывает влияния на выполняемые программы.

Описанный сценарий — один из примеров того, как NTFS использует файл журнала не только для восстановления файловой системы, но и для исправления ошибок при нормальной работе.

Восстановление

NTFS автоматически выполняет восстановление диска при первом обращении к нему какой-либо программы после загрузки системы. (Если восстановление не требуется, весь процесс тривиален.) При восстановлении используются две таблицы, которые NTFS поддерживает в памяти.

- *Таблица транзакций* (transaction table) — предназначена для отслеживания начатых, но еще не зафиксированных транзакций. В процессе восстановления результаты подопераций этих транзакций должны быть удалены с диска.
- *Таблица измененных страниц* (dirty page table) — в нее записывается информация о том, какие страницы кэша содержат изменения структуры файловой системы, еще не записанные на диск. Эти данные в процессе восстановления должны быть сброшены на диск.

Каждые 5 секунд NTFS добавляет в файл журнала транзакций запись контрольной точки. Непосредственно перед этим она обращается к LFS для сохранения в журнале текущей копии таблицы транзакций и таблицы измененных страниц. Затем NTFS запоминает в записи контрольной точки LSN записей журнала, содержащих копии таблиц. В начале процесса восстановления

после сбоя NTFS обращается к LFS для поиска записей журнала транзакций, содержащих самую последнюю запись контрольной точки и самые последние копии упомянутых выше таблиц. Затем NTFS копирует эти таблицы в память.

Обычно за последней записью контрольной точки в журнале транзакций находится еще несколько записей модификации. Эти записи соответствуют обновлениям тома, произошедшим после того, как запись контрольной точки была помещена в журнал. NTFS должна обновить таблицу транзакций и таблицу измененных страниц, включив в них информацию из этих дополнительных записей. После обновления таблиц NTFS использует их, а также содержимое журнала транзакций для обновления собственно тома.

При восстановлении тома NTFS выполняет три прохода по журналу транзакций, загружая его в память на первом проходе, чтобы минимизировать объем дискового ввода-вывода. Каждый проход имеет свое назначение:

- анализ;
- повтор транзакций;
- отмена транзакций.

Проход анализа

При *проходе анализа* (analysis pass) NTFS просматривает журнал транзакций в прямом направлении, начиная с последней операции контрольной точки, чтобы найти записи модификации и обновить скопированные ранее в память таблицы транзакций и измененных страниц. Обратите внимание, что операция контрольной точки помещает в журнал транзакций три записи, между которыми могут оказаться записи модификации (рис. 12-50). NTFS должна приступить к сканированию с начала операции контрольной точки.

Рис. 12-50. *Проход анализа*

Большинство записей модификации, расположенных в журнале после начала операции контрольной точки, представляет собой изменение либо таблицы транзакций, либо таблицы измененных страниц. Например, если запись модификации относится к фиксации транзакции, то транзакция, которую представляет данная запись, должна быть удалена из таблицы транзакций. Аналогично, если запись модификации относится к обновлению страницы, изменяющему структуру данных файловой системы, нужно внести соответствующую поправку в таблицу измененных страниц.

После того как таблицы в памяти приведены в актуальное состояние, NTFS просматривает их, чтобы определить LSN самой старой записи моди-

фикации, которая регистрирует операцию, не выполненную над диском. Таблица транзакций содержит LSN незафиксированных (незавершенных) транзакций, а таблица измененных страниц — LSN записей, соответствующих модификациям кэша, не отраженным на диске. LSN самой старой записи, найденной NTFS в этих двух таблицах, определяет, откуда начнется проход повтора. Однако, если последняя запись контрольной точки окажется более ранней, NTFS начнет проход повтора именно с нее.

Проход повтора

На *проходе повтора* (redo pass) NTFS сканирует журнал транзакций в прямом направлении, начиная с LSN самой старой записи, которая была обнаружена на проходе анализа (рис. 12-51). Она ищет записи модификации, относящиеся к обновлению страницы и содержащие модификации тома, которые были запротоколированы до сбоя системы, но не сброшены из кэша на диск. NTFS повторяет эти обновления в кэше.

Рис. 12-51. *Проход повтора*

Когда NTFS достигает конца журнала транзакций, она уже обновила кэш необходимыми модификациями тома, и подсистема отложенной записи, принадлежащая диспетчеру кэша, может начать переписывать содержимое кэша на диск в фоновом режиме.

Проход отмены

Завершив проход повтора, NTFS начинает *проход отмены* (undo pass), откатывая транзакции, не зафиксированные к моменту сбоя системы. На рис. 12-52 показаны две транзакции в журнале: транзакция 1 зафиксирована до сбоя системы, а транзакция 2 — нет. NTFS должна отменить транзакцию 2.

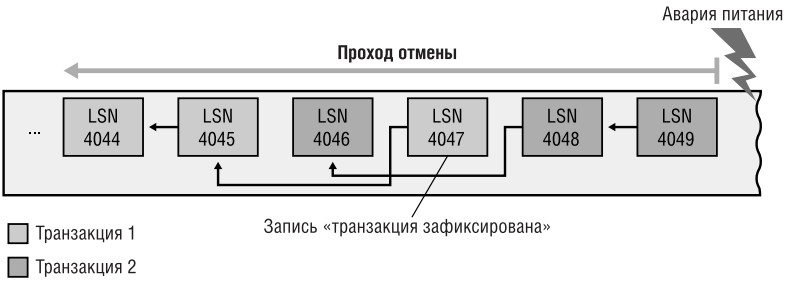


Рис. 12-52. *Проход отмены*

Допустим, в ходе транзакции 2 создавался файл. Эта операция состоит из трех подопераций (каждая со своей записью модификации). Записи модификации, относящиеся к одной транзакции, связываются в журнале обратными указателями, поскольку эти записи обычно не располагаются одна за другой.

В таблице транзакций NTFS для каждой незавершенной транзакции хранится LSN записи модификации, помещенной в журнал последней. В данном примере таблица транзакций сообщает, что для транзакции 2 это запись с LSN 4049. NTFS выполняет откат транзакции 2, как показано на рис. 12-53 (справа налево).

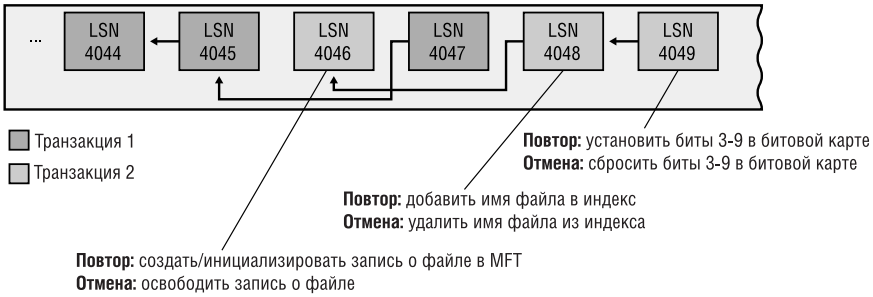


Рис. 12-53. *Отмена транзакции*

Найдя LSN 4049, NTFS извлекает информацию для отмены и выполняет отмену, сбрасывая биты 3–9 в своей битовой карте. Затем NTFS следует по обратному указателю к LSN 4048, который указывает ей удалить новое имя файла из индекса имен файлов. Наконец, NTFS переходит по последнему обратному указателю и освобождает запись MFT, зарезервированную для данного файла, в соответствии с информацией из записи модификации с LSN 4046. На этом откат транзакции 2 закончен. Если имеются другие незавершенные транзакции, NTFS повторяет ту же процедуру для их отката. Поскольку отмена транзакций изменяет структуру файловой системы на томе, NTFS должна протоколировать операцию отмены в журнале транзакций. В конце концов, при восстановлении может снова произойти сбой питания, и NTFS придется выполнить повтор операций отмены!

Когда проход отмены завершен, том возвращается в согласованное состояние. В этот момент NTFS сбрасывает на диск изменения кэша, чтобы гарантировать правильность содержимого тома. Далее NTFS записывает «пустую» область перезапуска, указывающую, что том находится в согласованном состоянии и что, если система сразу потерпит еще одну аварию, никакого восстановления не потребуется. На этом восстановление заканчивается.

NTFS гарантирует, что восстановление вернет том в некое существовавшее ранее целостное состояние, но не обязательно в непосредственно предшествовавшее сбою. NTFS не может дать такой гарантии, поскольку для большей производительности она использует алгоритм отложенной фиксации (lazy commit), а значит, сброс журнала транзакций из кэша на диск не выполняется немедленно всякий раз, когда добавляется запись «транзакция зафиксиро-

вана». Вместо этого несколько записей фиксации транзакций объединяются в пакет и записываются совместно — либо когда диспетчер кэша вызывает LFS для сброса журнала на диск, либо когда LFS помещает в журнал новую запись контрольной точки (каждые 5 секунд). Другая причина, по которой том не всегда возвращается к самому последнему состоянию, — в момент сбоя системы могли быть активны несколько параллельных транзакций, и одни записи фиксации этих транзакций были перенесены на диск, а другие — нет. Согласованное состояние тома, полученное в результате восстановления, отражает только те транзакции, чьи записи фиксации успели попасть на диск.

NTFS применяет журнал транзакций не только для восстановления тома, но и для других целей, реализация которых становится возможной за счет протоколирования транзакций. Файловые системы обязательно включают большой объем кода для обработки ошибок, возникающих в процессе обычного файлового ввода-вывода. Поскольку NTFS протоколирует каждую транзакцию, модифицирующую структуру тома, она может использовать журнал транзакций для восстановления после ошибок файловой системы и таким образом существенно упростить код обработки ошибок. Ошибка переполнения журнала транзакций, описанная ранее, — один из примеров использования протоколирования транзакций для обработки ошибок.

Большинство ошибок ввода-вывода, получаемых программой, не является ошибками файловой системы, и поэтому NTFS не может исправить их самостоятельно. Например, получив запрос на создание файла, NTFS может начать с создания записи в MFT, после чего ввести имя файла в индекс каталога. Однако при попытке выделить по своей битовой карте пространство для нового файла она может обнаружить, что диск заполнен, и запрос на создание файла удовлетворить не удастся. Тогда NTFS использует информацию из журнала транзакций для отмены уже выполненной части операции и освобождения структур данных, зарезервированных ею для файла. Затем ошибка «диск заполнен» возвращается вызывающей программе, которая и должна предпринять соответствующие действия.

Восстановление плохих кластеров в NTFS

Диспетчеры томов Windows — FtDisk (для базовых дисков) и LDM (для динамических) — могут восстанавливать данные из плохого (аварийного) сектора на отказоустойчивом томе, но, если жесткий диск не является SCSI-диском или если на нем больше нет резервных секторов, они не в состоянии заменить плохой сектор новым (подробнее о диспетчерах томов см. главу 10). В таком случае, когда файловая система считывает данные из плохого сектора, диспетчер томов восстанавливает информацию и возвращает ее вместе с соответствующим предупреждением.

Файловая система FAT никак не обрабатывает это предупреждение. Более того, ни эта файловая система, ни диспетчеры томов не ведут учет плохих секторов, поэтому, чтобы диспетчер томов не повторял все время восстановление данных из плохого сектора, пользователь должен запустить утилиту Chkdsk или Format. Обе эти утилиты далеко не идеальны в исключении пло-

хого сектора из дальнейшего использования. Chkdsk требует много времени для поиска и удаления плохих секторов, а Format уничтожает все данные в форматируемом разделе.

NTFS-эквивалент механизма замены секторов динамически заменяет кластер, содержащий плохой сектор, и ведет учет плохих кластеров, чтобы предотвратить их дальнейшее использование. (Вспомните, что NTFS адресуется к логическим кластерам, а не к физическим секторам.) Эта функциональность NTFS активизируется, если диспетчер томов не может заменить плохой сектор. Когда FtDisk возвращает предупреждение о плохом секторе или когда драйвер диска сообщает об ошибке, связанной с плохим сектором, NTFS выделяет новый кластер для замены того, который содержит плохой сектор. NTFS копирует данные, восстановленные диспетчером томов, в новый кластер, чтобы снова добиться избыточности данных.

На рис. 12-54 показана запись MFT для пользовательского файла, в одном из групп которого имеется плохой сектор. Когда NTFS получает ошибку, связанную с плохим сектором, она присоединяет содержащий его кластер к своему файлу плохих кластеров. Это предотвращает повторное выделение данного кластера другому файлу. Затем NTFS выделяет новый кластер и изменяет сопоставление VCN-LCN для файла так, чтобы оно указывало на этот кластер. Данная процедура, известная как *переназначение плохого кластера* (bad-cluster remapping), иллюстрируется на рис. 12-55. Кластер номер 1357, содержащий плохой сектор, заменяется новым кластером с номером 1049.

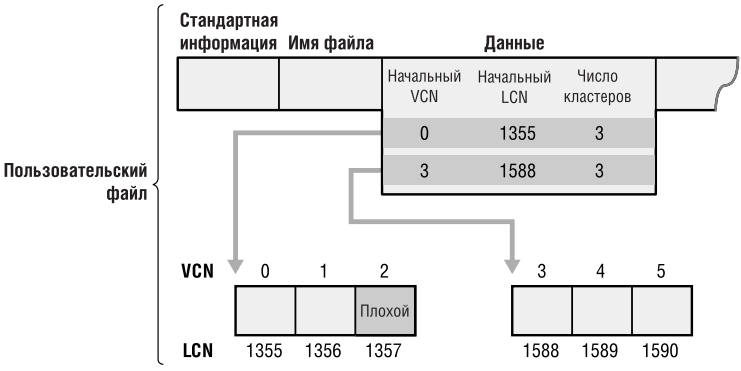


Рис. 12-54. *Запись MFT для пользовательского файла с плохим кластером*

Появление плохих секторов нежелательно, но когда они все же появляются, лучшее решение предлагается NTFS в сочетании с диспетчером томов. Если плохой сектор находится на томе с избыточностью данных, диспетчер томов восстанавливает данные и замещает сектор, если это возможно. Если сектор заменить нельзя, диспетчер томов возвращает предупреждение NTFS, которая заменяет кластер с плохим сектором.

Если том не сконфигурирован как отказоустойчивый, данные из плохого сектора восстановить нельзя. Когда том отформатирован для FAT и диспетчер томов не может восстановить данные, чтение из плохого сектора дает непред-

сказуемые результаты. Если в плохом секторе располагались какие-либо управляющие структуры файловой системы, может быть потерян целый файл или группа файлов (а иногда и весь диск). В лучшем случае будет потеряна часть данных файла (как правило, все файловые данные, расположенные в данном секторе и за ним). Более того, FAT скорее всего повторно выделит плохой сектор под другой файл, в результате чего проблема возникнет снова.

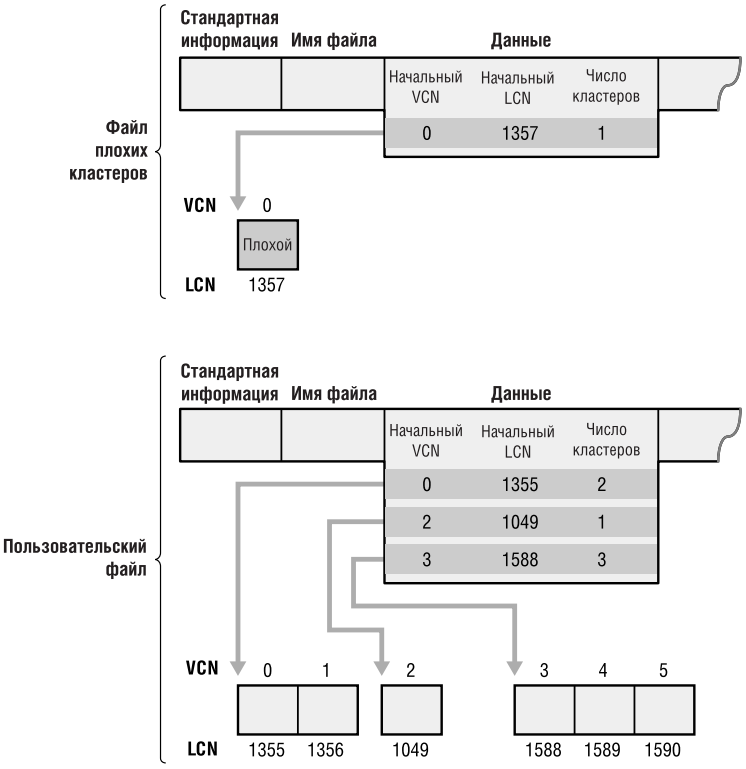


Рис. 12-55. *Переназначение плохих кластеров*

Как и другие файловые системы, NTFS не может восстановить данные из плохого сектора без помощи диспетчера томов. Однако она значительно сокращает ущерб, который наносится появлением плохого сектора. Если NTFS обнаруживает плохой сектор в ходе операции чтения, она переназначает кластер, как показано на рис. 12-55. Если том не сконфигурирован для избыточного хранения информации, NTFS возвращает ошибку чтения вызывающей программе. Хотя данные, находившиеся в этом кластере, теряются, остаток файла и сама файловая система сохраняются, вызывающая программа может соответствующим образом отреагировать на потерю данных, а плохой кластер больше не будет использоваться при распределении пространства на томе. Если NTFS обнаруживает плохой кластер во время запи-

си, а не чтения, она переназначает его до выполнения операции записи и тем самым вообще избегает потери данных.

Та же процедура восстановления используется, когда в аварийном секторе хранятся данные файловой системы. Если он находится на томе с избыточностью данных, NTFS динамически заменяет соответствующий кластер, используя данные, восстановленные диспетчером томов. Если том не избыточный, восстановить данные нельзя, и NTFS устанавливает в файле тома бит, указывающий, что том поврежден. При перезагрузке системы NTFS-утилита Chkdsk проверяет этот бит и, если он установлен, исправляет повреждение файловой системы, реконструируя метаданные NTFS.

Крайне редко повреждение файловой системы может произойти даже на отказоустойчивом томе: двойная ошибка способна разрушить и данные файловой системы, и информацию для их восстановления. Если авария системы происходит в тот момент, когда NTFS сохраняет, например, зеркальную копию записи MFT, индекса имен файлов или журнала транзакций, то зеркальная копия этих данных файловой системы обновляется не полностью. Если после перезагрузки системы ошибка, связанная с плохим сектором, возникает в том же месте основного диска, где находится частично записанная зеркальная копия, NTFS не может восстановить информацию с зеркального диска. Для детекции таких повреждений системных данных в NTFS реализована специальная схема. Если обнаружено нарушение целостности, в файле тома устанавливается бит повреждения, в результате чего при следующей загрузке системы метаданные NTFS будут реконструированы Chkdsk. Так как в отказоустойчивой дисковой конфигурации повреждение данных файловой системы маловероятно, потребность в Chkdsk возникает редко. Эта утилита позиционируется как дополнительная мера предосторожности, а не как основное средство восстановления информации.

Использование Chkdsk в NTFS существенно отличается от ее использования в FAT. Перед записью на диск FAT устанавливает бит изменения тома, который сбрасывается по завершении модификации тома. Если сбой системы происходит при выполнении операции вывода, бит остается установленным, и после перезагрузки машины запускается Chkdsk. В NTFS утилита Chkdsk запускается, только когда обнаруживаются неожиданные или нечитаемые данные файловой системы и NTFS не может восстановить их с избыточного тома или из избыточных структур данных файловой системы на обычном томе. (Система дублирует загрузочный сектор, равно как и части MFT, необходимые для загрузки системы и выполнения процедуры восстановления NTFS. Эта избыточность гарантирует, что NTFS всегда сможет загрузиться и восстановить сама себя.)

В таблицу 12-6 сведены данные о том, что происходит при появлении плохого сектора на дисковом томе (отформатированном для одной из файловых систем Windows) в различных ситуациях, описанных в данном разделе.

Таблица 12-6. *Сценарии восстановления данных NTFS*

Сценарий	При наличии SCSI-диска, имеющего резервные секторы	При наличии диска, отличного от SCSI, или SCSI-диска без резервных секторов[1]
Отказоустойчивый том[2]	1. Диспетчер томов восстанавливает данные	1. Диспетчер томов восстанавливает данные
	2. Диспетчер томов выполняет замену сектора (заменяет плохой сектор)	2. Диспетчер томов возвращает файловой системе данные и ошибку «плохой сектор»
	3. Файловой системе не сообщается об ошибке	3. NTFS переназначает кластер
Обычный том	1. Диспетчер томов не может восстановить данные	1. Диспетчер томов не может восстановить данные
	2. Диспетчер томов возвращает файловой системе ошибку «плохой сектор»	2. Диспетчер томов возвращает файловой системе ошибку «плохой сектор»
	3. NTFS переназначает кластер; данные теряются[3]	3. NTFS переназначает кластер; данные теряются[3]

[1] Диспетчер томов не может заменять секторы ни в одном из следующих случаев: 1) жесткие диски, отличные от SCSI, не поддерживают стандартный интерфейс для замены секторов; 2) некоторые жесткие диски не имеют аппаратной поддержки замены секторов, а SCSI-диски, у которых она есть, могут со временем исчерпать весь резерв секторов.

[2] Отказоустойчивым является том одного из следующих типов: зеркальный или RAID-5.

[3] При записи данные не теряются: NTFS переназначает кластер до выполнения записи.

Если том, на котором появился плохой сектор, сконфигурирован как отказоустойчивый, а жесткий диск поддерживает замену секторов и его запас резервных секторов еще не исчерпан, то тип файловой системы (FAT или NTFS) не имеет значения. Диспетчер томов заменяет плохой сектор, не требуя вмешательства со стороны пользователя или файловой системы.

Если плохой сектор появился на жестком диске, не поддерживающем замену секторов, то за переназначение плохого сектора или, как в случае NTFS, кластера, в котором находится плохой сектор, отвечает файловая система. FAT не умеет переназначать секторы или кластеры.

Механизм EFS

EFS (Encrypting File System) использует средства поддержки шифрования. При первом шифровании файла EFS назначает учетной записи пользователя, шифрующего этот файл, криптографическую пару — закрытый и открытый ключи. Пользователи могут шифровать файлы с помощью Windows Explorer; для этого нужно открыть диалоговое окно Properties (Свойства) применительно к нужному файлу, щелкнуть кнопку Advanced (Другие) и установить флажок

Encrypt Contents To Secure Data (Шифровать содержимое для защиты данных), как показано на рис. 12-56. Пользователи также могут шифровать файлы с помощью утилиты командной строки *cipher*. Windows автоматически шифрует файлы в каталогах, помеченных зашифрованными. При шифровании файла EFS генерирует случайное число, называемое шифровальным ключом файла (file encryption key, FEK). EFS использует FEK для шифрования содержимого файла по более стойкому варианту DES (Data Encryption Standard) — DESX (в Windows 2000), а также по DESX, 3DES (Triple-DES) или AES (Advanced Encryption Standard) в Windows XP (Service Pack 1 и выше) и Windows Server 2003. EFS сохраняет FEK вместе с самим файлом, но FEK шифруется по алгоритму RSA-шифрования на основе открытого ключа. После выполнения EFS этих действий файл защищен: другие пользователи не смогут расшифровать данные без расшифрованного FEK файла, а FEK они не смогут расшифровать без закрытого ключа пользователя — владельца файла.

Стойкость алгоритмов шифрования FEK

По умолчанию FEK шифруется в Windows 2000 и Windows XP по алгоритму DESX, а в Windows XP с Service Pack 1 (или выше) и Windows Server 2003 — по алгоритму AES. В версиях Windows, разрешенных к экспорту за пределы США, драйвер EFS реализует 56-битный ключ шифрования DESX, тогда как в версии, подлежащей использованию только в США, и в версиях с пакетом для 128-битного шифрования длина ключа DESX равна 128 битам. Алгоритм AES в Windows использует 256-битные ключи. Применение 3DES разрешает доступ к более длинным ключам, поэтому, если вам требуется более высокая стойкость FEK, вы можете включить шифрование 3DES одним из двух способов: как алгоритм шифрования для всех криптографических сервисов в системе или только для EFS.

Чтобы 3DES стал алгоритмом шифрования для всех системных криптографических сервисов, запустите редактор локальной политики безопасности, введя **secpol.msc** в диалоговом окне Run (Запуск программы), и откройте узел Security Options (Параметры безопасности) под Local Policies (Локальные политики). Найдите параметр System Cryptography: Use FIPS Compliant Algorithms For Encryption, Hashing And Signing (Системная криптография: использовать FIPS-совместимые алгоритмы для шифрования, хеширования и подписывания) и включите его.

Чтобы активизировать 3DES только для EFS, создайте DWORD-параметр HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\EFS\AlgorithmID, присвойте ему значение 0x6603 и перезагрузите систему.

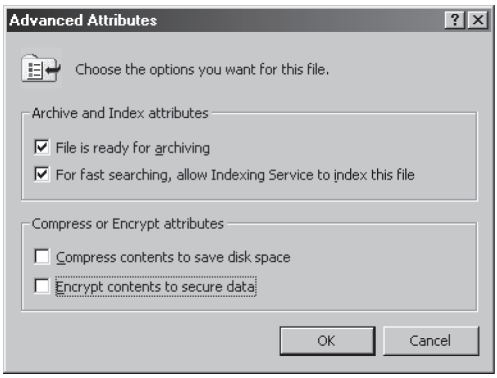


Рис. 12-56. *Шифрование файлов через пользовательский интерфейс Windows Explorer*

Для шифрования FEK используется алгоритм криптографической пары, а для шифрования файловых данных — DESX, AES или 3DES (все это алгоритмы симметричного шифрования, в которых применяется один и тот же ключ для шифрования и дешифрования). Как правило, алгоритмы симметричного шифрования работают очень быстро, что делает их подходящими для шифрования больших объемов данных, в частности файловых. Однако у алгоритмов симметричного шифрования есть одна слабая сторона: зашифрованный ими файл можно вскрыть, получив ключ. Если несколько человек собирается пользоваться одним файлом, защищенным только DESX, AES или 3DES, каждому из них понадобится доступ к FEK файла. Очевидно, что незашифрованный FEK — серьезная угроза безопасности. Но шифрование FEK все равно не решает проблему, поскольку в этом случае нескольким людям приходится пользоваться одним и тем же ключом расшифровки FEK.

Защита FEK — сложная проблема, для решения которой EFS использует ту часть своей криптографической архитектуры, которая опирается на технологии шифрования с открытым ключом. Шифрование FEK на индивидуальной основе позволяет нескольким лицам совместно использовать зашифрованный файл. EFS может зашифровать FEK файла с помощью открытого ключа каждого пользователя и хранить их FEK вместе с файлом. Каждый может получить доступ к открытому ключу пользователя, но никто не сможет расшифровать с его помощью данные, зашифрованные по этому ключу. Единственный способ расшифровки файла заключается в использовании операционной системой закрытого ключа. Закрытый ключ помогает расшифровать нужный зашифрованный экземпляр FEK файла. Алгоритмы на основе открытого ключа обычно довольно медленные, поэтому они используются EFS только для шифрования FEK. Разделение ключей на открытый и закрытый немного упрощает управление ключами по сравнению с таковым в алгоритмах симметричного шифрования и решает дилемму, связанную с защитой FEK.

Windows хранит закрытые ключи в подкаталоге Application Data\Microsoft\Crypto\RSA каталога профиля пользователя. Для защиты закрытых клю-

чей Windows шифрует все файлы в папке RSA на основе симметричного ключа, генерируемого случайным образом; такой ключ называется *мастер-ключом* пользователя. Мастер-ключ имеет длину в 64 байта и создается стойким генератором случайных чисел. Мастер-ключ также хранится в профиле пользователя в каталоге Application Data\Microsoft\Protect и зашифровывается по алгоритму 3DES с помощью ключа, который отчасти основан на пароле пользователя. Когда пользователь меняет свой пароль, мастер-ключи автоматически расшифровываются, а затем заново зашифровываются с учетом нового пароля.

Функциональность EFS опирается на несколько компонентов, как видно на схеме архитектуры EFS (рис. 12-57). В Windows 2000 EFS реализована в виде драйвера устройства, работающего в режиме ядра и тесно связанного с драйвером файловой системы NTFS, но в Windows XP и Windows Server 2003 поддержка EFS включена в драйвер NTFS. Всякий раз, когда NTFS встречает шифрованный файл, она вызывает функции EFS, зарегистрированные кодом EFS режима ядра в NTFS при инициализации этого кода. Функции EFS осуществляют шифрование и расшифровку файловых данных по мере обращения приложений к шифрованным файлам. Хотя EFS хранит FEK вместе с данными файла, FEK шифруется с помощью открытого ключа индивидуального пользователя. Для шифрования или расшифровки файловых данных EFS должна расшифровать FEK файла, обращаясь к криптографическим сервисам пользовательского режима.

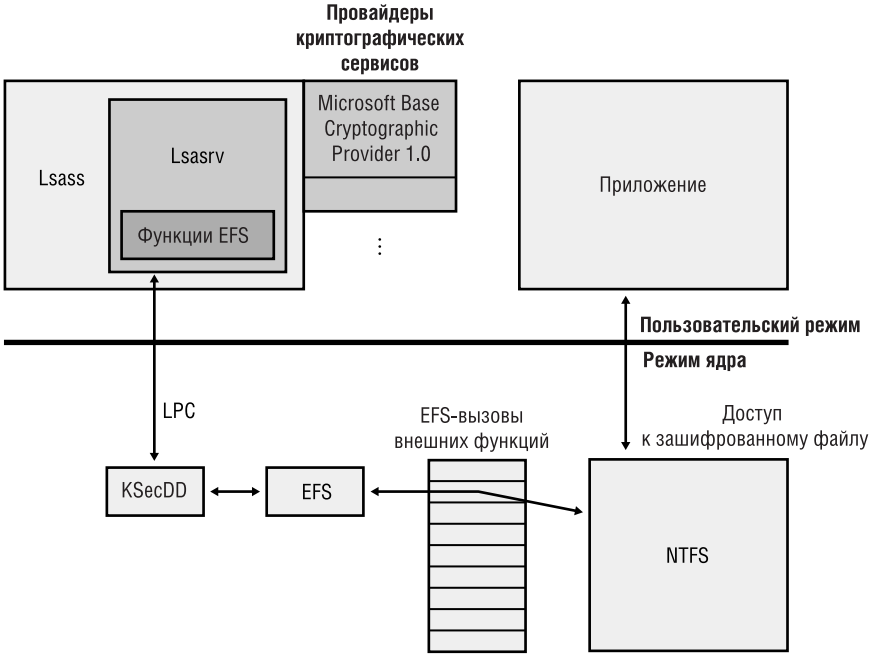


Рис. 12-57. *Архитектура EFS*

Подсистема локальной аутентификации (Local Security Authentication Subsystem, LSASS) (\Windows\System32\Lsass.exe) не только управляет сеансами регистрации, но и выполняет все рутинные операции, связанные с управлением ключами EFS. Например, когда драйверу EFS требуется расшифровать FEK для расшифровки данных файла, к которому обращается пользователь, драйвер EFS посылает запрос LSASS через LPC. Драйвер устройства KSecDD (\Windows\System32\Drivers\Ksecdd.sys) экспортирует функции, необходимые драйверам для посылки LPC-сообщений LSASS. Компонент LSASS, сервер локальной аутентификации (Local Security Authentication Server, Lsasrv) (\Windows\System32\Lsasrv.dll), ожидает запросы на расшифровку FEK через RPC; расшифровка выполняется соответствующей функцией EFS, которая также находится в Lsasrv. Для расшифровки FEK, передаваемого LSASS драйвером EFS в зашифрованном виде, Lsasrv использует функции Microsoft CryptoAPI (CAPI).

CryptoAPI состоит из DLL провайдеров криптографических сервисов (cryptographic service providers, CSP), которые обеспечивают приложениям доступ к различным криптографическим сервисам (шифрованию, дешифрованию и хэшированию). Например, эти DLL управляют получением открытого и закрытого ключей пользователя, что позволяет Lsasrv не заботиться о деталях защиты ключей и даже об особенностях работы алгоритмов шифрования. EFS опирается на алгоритмы шифрования RSA, предоставляемые провайдером Microsoft Enhanced Cryptographic Provider (\Windows\System32\Rsaenh.dll). Расшифровав FEK, Lsasrv возвращает его драйверу EFS в ответном LPC-сообщении. Получив расшифрованный FEK, EFS с помощью DESX расшифровывает данные файла для NTFS.

Подробнее о том, как EFS взаимодействует с NTFS и как Lsasrv управляет ключами через CryptoAPI, мы поговорим в следующих разделах.

Первое шифрование файла

Обнаружив шифрованный файл, драйвер NTFS вызывает функции EFS. О состоянии шифрования файла сообщают его атрибуты — так же, как и о состоянии сжатия в случае сжатых файлов. NTFS и EFS имеют специальные интерфейсы для преобразования файла из незашифрованной в зашифрованную форму, но этот процесс протекает в основном под управлением компонентов пользовательского режима. Как уже говорилось, Windows позволяет шифровать файлы двумя способами: утилитой командной строки *cipher* или установкой флажка Encrypt Contents To Secure Data на вкладке Advanced Attributes окна свойств файла в Windows Explorer. Windows Explorer и *cipher* используют Windows-функцию *EncryptFile*, экспортируемую Advapi32.dll (Advanced Windows API DLL). Чтобы получить доступ к API, который нужен для LPC-вызова интерфейсов EFS в Lsasrv, Advapi32 загружает другую DLL, Feclient.dll (File Encryption Client DLL).

Получив RPC-сообщение с запросом на шифрование файла от Feclient, Lsasrv использует механизм олицетворения Windows для подмены собой пользователя, запустившего программу, шифрующую файл (*cipher* или Win-

dows Explorer). Это заставляет Windows воспринимать файловые операции, выполняемые Lsasrv, как операции, выполняемые пользователем, желающим зашифровать файл. Lsasrv обычно работает под учетной записью System (об этой учетной записи см. главу 8). Если бы Lsasrv не олицетворял пользователя, то не получил бы прав на доступ к шифруемому файлу.

Далее Lsasrv создает файл журнала в каталоге System Volume Information, где регистрирует ход процесса шифрования. Имя файла журнала — обычно Efs0.log, но, если шифруется несколько файлов, 0 заменяется числом, которое последовательно увеличивается на 1 до тех пор, пока не будет получено уникальное имя журнала для текущего шифруемого файла.

CryptoAPI полагается на информацию пользовательского профиля, хранящуюся в реестре, поэтому, если профиль еще не загружен, следующий шаг Lsasrv — загрузка в реестр профиля олицетворяемого пользователя вызовом функции *LoadUserProfile* из Userenv.dll (User Environment DLL). Обычно профиль пользователя к этому моменту уже загружен, поскольку Winlogon загружает его при входе пользователя в систему. Но если пользователь регистрируется под другой учетной записью с помощью команды *RunAs*, то при попытке обращения к зашифрованному файлу под этой учетной записью соответствующий профиль может быть не загружен.

После этого Lsasrv генерирует для файла FEK, обращаясь к средствам шифрования RSA, реализованным в Microsoft Base Cryptographic Provider 1.0.

Создание связок ключей

К этому моменту Lsasrv уже получил FEK и может сгенерировать информацию EFS, сохраняемую вместе с файлом, включая зашифрованную версию FEK. Lsasrv считывает из параметра реестра HKCU\Software\Microsoft\Windows NT\CurrentVersion\EFS\CurrentKeys\CertificateHash значение, присвоенное пользователю, который затребовал операцию шифрования, и получает сигнатуру открытого ключа этого пользователя. (Заметьте, что этот раздел не появляется в реестре, если ни один файл или каталог не зашифрован.) Lsasrv использует эту сигнатуру для доступа к открытому ключу пользователя и для шифрования FEK.

Теперь Lsasrv может создать информацию, которую EFS сохранит вместе с файлом. EFS хранит в шифрованном файле только один блок информации, в котором содержатся записи для всех пользователей этого файла. Данные записи называются *элементами ключей* (key entries); они хранятся в области сопоставленных с файлом данных EFS, которая называется Data Decryption Field (DDF). Совокупность нескольких элементов ключей называется *связкой ключей* (key ring), поскольку, как уже говорилось, EFS позволяет нескольким лицам совместно использовать шифрованный файл.

Формат данных EFS, сопоставленных с файлом, и формат элемента ключа показан на рис. 12-58. В первой части элемента ключа EFS хранит информацию, достаточную для точного описания открытого ключа пользователя. В нее входит SID пользователя (его наличие не гарантируется), имя контейнера, в котором хранится ключ, имя провайдера криптографических серви-

сов и хэш сертификата криптографической пары (при расшифровке используется только этот хэш). Во второй части элемента ключа содержится шифрованная версия FEK. Lsasrv шифрует FEK через CryptoAPI по алгоритму RSA с применением открытого ключа данного пользователя.

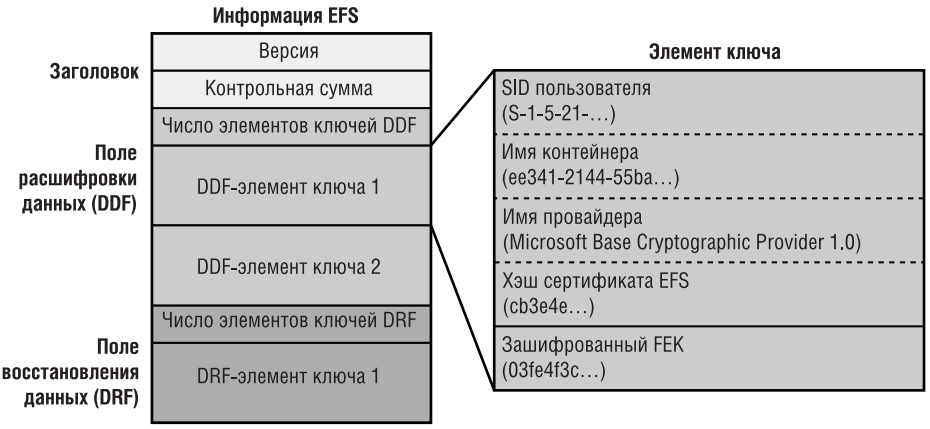


Рис. 12-58. *Формат информации EFS и элемента ключа*

Далее Lsasrv создает еще одну связку ключей, содержащую элементы ключей восстановления (recovery key entries). EFS хранит информацию об этих элементах в поле DRF файла (см. рис. 12-58). Формат элементов DRF идентичен формату DDF. DRF служит для расшифровки пользовательских данных по определенным учетным записям (агентов восстановления) в тех случаях, когда администратору нужен доступ к пользовательским данным. Допустим, сотрудник компании забыл свой пароль для входа в систему. В этом случае администратор может сбросить пароль этого сотрудника, но без агентов восстановления никто не сумеет восстановить его зашифрованные данные.

Агенты восстановления (Recovery Agents) определяются в политике безопасности Encrypted Data Recovery Agents (Агенты восстановления шифрованных данных) на локальном компьютере или в домене. Эта политика доступна через оснастку Group Policy (Групповая политика) консоли MMC. Запустите Recovery Agent Wizard (Мастер добавления агента восстановления), щелкнув правой кнопкой мыши строку Encrypted Data Recovery Agents (рис. 12-59) и последовательно выбрав команды New (Создать) и Encrypted Recovery Agent (Агент восстановления шифрованных данных). Вы можете добавить агенты восстановления и указать, какие криптографические пары (обозначенные их сертификатами) могут использовать эти агенты для восстановления шифрованных данных. Lsasrv интерпретирует политику восстановления в процессе своей инициализации или при получении уведомления об изменении политики восстановления. EFS создает DRF-элементы ключей для каждого агента восстановления, используя провайдер криптографических сервисов, зарегистрированный для EFS-восстановления. Провайдером по умолчанию служит Base Cryptographic Provider 1.0.

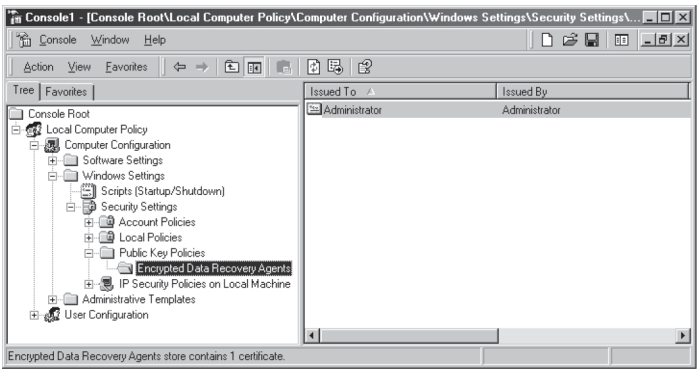


Рис. 12-59. *Групповая политика Encrypted Data Recovery Agents (Агенты восстановления шифрованных данных)*

На завершающем этапе создания информации EFS для файла Lsasrv вычисляет контрольную сумму для DDF и DRF по механизму хэширования MD5 из Base Cryptographic Provider 1.0. Lsasrv хранит вычисленную контрольную сумму в заголовке данных EFS. EFS ссылается на эту сумму при расшифровке, чтобы убедиться в том, что сопоставленные с файлом данные EFS не повреждены и не взломаны.

Шифрование файловых данных

Ход процесса шифрования показан на рис. 12-60. После создания всех данных, необходимых для шифруемого пользователем файла, Lsasrv приступает к шифрованию файла и создает его резервную копию, Efs0.tmp (если есть другие резервные копии, Lsasrv просто увеличивает номер в имени резервного файла). Резервная копия помещается в тот каталог, где находится шифруемый файл. Lsasrv применяет к резервной копии ограничивающий дескриптор защиты, так что доступ к этому файлу можно получить только по учетной записи System. Далее Lsasrv инициализирует файл журнала, созданный им на первом этапе процесса шифрования и регистрирует в нем факт создания резервного файла. Lsasrv шифрует исходный файл только после его резервирования.

Наконец, Lsasrv посылает через NTFS коду EFS режима ядра команду на добавление к исходному файлу созданной информации EFS. В Windows 2000 NTFS получает эту команду, но поскольку она не понимает команд EFS, то просто вызывает драйвер EFS. Код EFS режима ядра принимает посланные Lsasrv данные EFS и применяет их к файлу через функции, экспортируемые NTFS. Эти функции позволяют EFS добавлять к NTFS-файлам атрибут $EFS. После этого управление возвращается к Lsasrv, который копирует содержимое шифруемого файла в резервный. Закончив создание резервной копии (в том числе скопировав все дополнительные потоки данных), Lsasrv отмечает в файле журнала, что резервный файл находится в актуальном состоянии. Затем Lsasrv посылает NTFS другую команду, требуя зашифровать содержимое исходного файла.

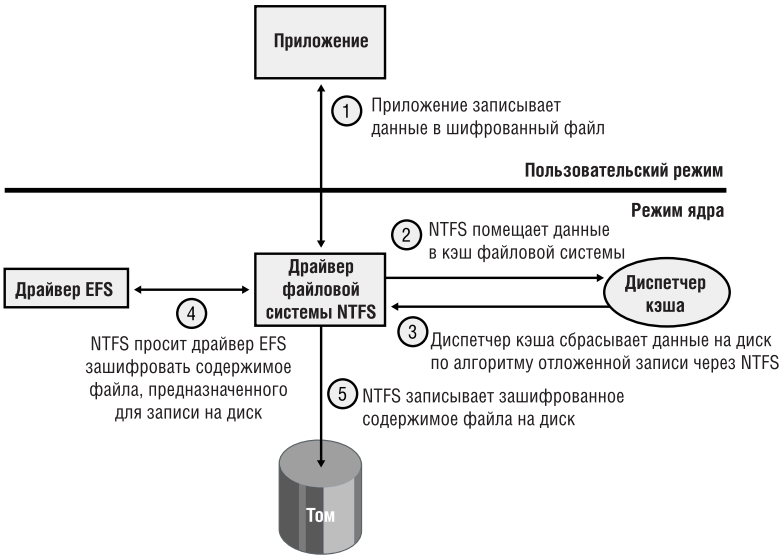


Рис. 12-60. *Схема работы EFS*

Получив от EFS команду на шифрование файла, NTFS удаляет содержимое исходного файла и копирует в него данные резервного. По мере копирования каждого раздела файла NTFS сбрасывает данные раздела из кэша файловой системы, и они записываются на диск. Поскольку файл помечен как шифрованный, NTFS вызывает EFS для шифрования раздела данных перед записью на диск. EFS использует незашифрованный FEK, переданный NTFS, чтобы шифровать файловые данные по алгоритму DESX, AES или 3DES порциями, равными по размеру одному сектору (512 байтов).

После того как файл зашифрован, Lsasrv регистрирует в файле журнала, что шифрование успешно завершено, и удаляет резервную копию файла. В заключение Lsasrv удаляет файл журнала и возвращает управление приложению, запросившему шифрование файла.

Сводная схема процесса шифрования

Ниже приведен сводный список этапов шифрования файла через EFS.

1. Загружается профиль пользователя, если это необходимо.
2. В каталоге System Volume Information создается файл журнала с именем Efs*x*.log, где *x* — уникальное целое число от 0. По мере выполнения следующих этапов в журнал заносятся записи, позволяющие восстановить файл после сбоя системы в процессе шифрования.
3. Base Cryptographic Provider 1.0 генерирует для файла случайное 128-битное число, используемое в качестве FEK.
4. Генерируется или считывается криптографическая пара ключей пользователя. Она идентифицируется в HKCU\Software\Microsoft\Windows NT\CurrentVersion\EFS\CurrentKeys\CertificateHash.

5. Для файла создается связка ключей DDF с элементом для данного пользователя. Этот элемент содержит копию FEK, зашифрованную с помощью открытого EFS-ключа пользователя.
6. Для файла создается связка ключей DRF. В нем есть элементы для каждого агента восстановления в системе, и при этом в каждом элементе содержится копия FEK, зашифрованная с помощью открытого EFS-ключа агента.
7. Создается резервный файл с именем вида Efs0.tmp в том каталоге, где находится и шифруемый файл.
8. Связки ключей DDF и DRF добавляются к заголовку и сопоставляются с файлом как атрибут EFS.
9. Резервный файл помечается как шифрованный, и в него копируется содержимое исходного файла.
10. Содержимое исходного файла уничтожается, в него копируется содержимое резервного. В результате этой операции данные исходного файла шифруются, так как теперь файл помечен как шифрованный.
11. Удаляется резервный файл.
12. Удаляется файл журнала.
13. Выгружается профиль пользователя (загруженный на этапе 1).

При сбое системы во время шифрования согласованные данные непременно сохранятся в одном из файлов — исходном или резервном. Когда Lsasrv инициализируется после сбоя системы, он ищет файлы журнала в каталоге System Volume Information на каждом NTFS-томе в системе. Если Lsasrv находит один или несколько файлов журнала, он изучает их содержимое и определяет порядок восстановления. Если исходный файл не был модифицирован на момент аварии, Lsasrv удаляет файл журнала и соответствующий резервный файл; в ином случае он копирует резервный файл поверх исходного (частично шифрованного) файла, после чего удаляет журнал и резервную копию. После того как Lsasrv обработает файлы журналов, файловая система возвращается в целостное состояние без потери пользовательских данных.

Процесс расшифровки

Процесс расшифровки начинается, когда пользователь открывает шифрованный файл. При открытии файла NTFS анализирует его атрибуты и выполняет функцию обратного вызова в драйвере EFS. Драйвер EFS считывает атрибут $EFS, сопоставленный с шифрованным файлом. Чтобы прочитать этот атрибут, драйвер вызывает функции поддержки EFS, которые NTFS экспортирует для EFS. NTFS выполняет все необходимые действия, чтобы открыть файл. Драйвер EFS проверяет наличие у пользователя, открывающего файл, прав доступа к данным шифрованного файла (т. е. зашифрованный FEK в связке ключей DDF или DRF должен соответствовать криптографической паре ключей, сопоставленной с пользователем). После такой проверки EFS получает расшифрованный FEK файла, применяемый для обработки данных в операциях, которые пользователь может выполнять над файлом.

EFS не может расшифровать FEK самостоятельно и полагается в этом на Lsasrv (который может использовать CryptoAPI). С помощью драйвера Ksecdd.sys EFS посылает LPC-сообщение Lsasrv, чтобы тот извлек из атрибута $EFS (т. е. из данных EFS) FEK пользователя, открывающего файл, и расшифровал его.

Получив LPC-сообщение, Lsasrv вызывает функцию *LoadUserProfile* из Userenv.dll для загрузки в реестр профиля пользователя, если он еще не загружен. Lsasrv перебирает все поля ключей в данных EFS, пробуя расшифровать каждый FEK на основе закрытого ключа пользователя; с этой целью Lsasrv пытается расшифровать FEK в DDF- или DRF-элементе ключа. Если хэш сертификата в поле ключа не подходит к ключу пользователя, Lsasrv переходит к следующему полю ключа. Если Lsasrv не удастся расшифровать ни одного FEK в DDF или DRF, пользователь не получит FEK файла, и EFS запретит доступ к файлу приложению, которое пыталось открыть этот файл. А если Lsasrv найдет какой-нибудь хэш, который соответствует ключу пользователя, он расшифрует FEK по закрытому ключу пользователя через CryptoAPI.

Lsasrv, обрабатывая при расшифровке FEK связки ключей DDF и DRF, автоматически выполняет операции восстановления файла. Если к файлу пытается получить доступ агент восстановления, не зарегистрированный на доступ к шифрованному файлу (т. е. у него нет соответствующего поля в связке ключей DDF), EFS позволит ему обратиться к файлу, потому что агент имеет доступ к паре ключей для поля ключа в связке ключей DRF.

Кэширование расшифрованного FEK

Путь от драйвера EFS до Lsasrv и обратно требует довольно много времени — в процессе расшифровки FEK в типичной системе CryptoAPI использует результаты более 2000 вызовов API-функций реестра и 400 обращений к файловой системе. Чтобы сократить издержки от всех этих вызовов, драйвер EFS использует кэш в паре с NTFS.

Расшифровка файловых данных

Открыв шифрованный файл, приложение может читать и записывать его данные. Для расшифровки файловых данных NTFS вызывает драйвер EFS по мере чтения этих данных с диска — до того, как помещает их в кэш файловой системы. Аналогичным образом, когда приложение записывает данные в файл, они остаются незашифрованными в кэше файловой системы, пока приложение или диспетчер кэша не сбросит данные обратно на диск с помощью NTFS. При записи данных шифрованного файла из кэша на диск NTFS вызывает драйвер EFS, чтобы зашифровать их.

Как уже говорилось, драйвер EFS выполняет шифрование и расшифровку данных порциями по 512 байтов. Такой размер оптимален для драйвера, потому что объем данных при операциях чтения и записи кратен размеру сектора.

Резервное копирование шифрованных файлов

Важный аспект разработки любого механизма шифрования файлов заключается в том, что приложения не могут получить доступ к расшифрованным данным иначе, чем через механизмы шифрования. Это ограничение особенно важно для утилит резервного копирования, с помощью которых файлы сохраняются на архивных носителях. EFS решает эту проблему, предоставляя утилитам резервного копирования механизм, с помощью которого они могут создавать резервные копии файлов и восстанавливать их в шифрованном виде. Таким образом, утилитам резервного копирования не обязательно шифровывать или расшифровывать данные файлов в процессе резервного копирования.

Для доступа к шифрованному содержимому файлов утилиты резервного копирования в Windows используют новый EFS API: функции *OpenEncryptedFileRaw*, *ReadEncryptedFileRaw*, *WriteEncryptedFileRaw* и *CloseEncryptedFileRaw*. Эти функции, предоставляемые Advapi32.dll, вызывают соответствующие функции Lsasrv по механизму LPC. Например, после того как утилита резервного копирования открывает файл, она вызывает *ReadEncryptedFileRaw*, чтобы получить данные. Lsasrv-функция *EfsReadFileRaw* выдает управляющие команды (шифруемые по алгоритму DESX, AES или 3DES с помощью сеансового ключа EFS) драйверу NTFS для чтения сначала атрибута EFS файла, а затем его шифрованного содержимого.

EfsReadFileRaw может понадобиться несколько операций чтения, чтобы считать большой файл. По мере того как *EfsReadFileRaw* считывает очередную порцию файла, Lsasrv посылает Advapi32.dll RPC-сообщение, в результате которого выполняется функция обратного вызова, указанная программой резервного копирования при вызове *ReadEncryptedFileRaw*. Функция *EfsReadFileRaw* передает считанные шифрованные данные функции обратного вызова, которая записывает их на архивный носитель. Восстанавливаются шифрованные файлы аналогичным образом. Программа резервного копирования вызывает API-функцию *WriteEncryptedFileRaw*, которая активизирует функцию обратного вызова программы резервного копирования для получения нешифрованных данных с архивного носителя, в то время как Lsasrv-функция *EfsWriteFileRaw* восстанавливает содержимое файла.

ЭКСПЕРИМЕНТ: просмотр информации EFS

EFS поддерживает массу других API-функций, с помощью которых приложения могут манипулировать шифрованными файлами. Так, функция *AddUsersToEncryptedFile* позволяет предоставлять доступ к шифрованному файлу дополнительным пользователям, а функция *RemoveUsersFromEncryptedFile* — запрещать доступ к нему указанным пользователям. Функция *QueryUsersOnEncryptedFile* сообщает о сопоставленных с файлом полях ключей DDF и DRF. Она возвращает SID, хэш сертификата и содержимое каждого поля ключа DDF и DRF. Ниже приведен образец вывода утилиты EFSDump (*www.sysinternals.com*).

см. след. стр.

В качестве параметра ее командной строки указано имя шифрованного файла.

```
C:\>efsdump test.txt
EFS Information Dumper v1.02
Copyright (C) 1999 Mark Russinovich
Systems Internals - http://www.sysinternals.com

test.txt:
DDF Entry:
    DARYL\Mark:
         CN=Mark,L=EFS,OU=EFS File Encryption Certificate
DRF Entry:
    DARYL\Administrator
         OU=EFS File Encryption Certificate, L=EFS, CN=Administrator
```

Как видите, в файле test.txt имеется один элемент DDF, соответствующий пользователю Mark, и один элемент DRF, соответствующий Administrator, который является единственным агентом восстановления, зарегистрированным в системе на данный момент.

Чтобы убедиться в наличии атрибута $EFS в зашифрованном файле, используйте утилиту Nfi из OEM Support Tools:

```
C:\>nfi test.txt
NTFS File Sector Information Utility.Copyright (C) Microsoft
Corporation 1999. All rights reserved.
\Temp\README.TXT    $STANDARD_INFORMATION (resident)    $FILE_NAME
(resident)    $DATA (nonresident)        logical sectors 2728124-2728135
(0x29a0bc-0x29a0c7)    Attribute Type 0x100 $EFS (nonresident)
logical sectors 2156984-2156987 (0x20e9b8-0x20e9bb)
```

Резюме

Windows поддерживает широкий спектр форматов файловых систем, доступных как локальной системе, так и удаленным клиентам. Архитектура драйвера фильтра файловой системы позволяет корректно расширять и дополнять средства доступа к файловой системе, а NTFS является надежным, безопасным и масштабируемым форматом файловой системы. В следующей главе мы рассмотрим поддержку сетей в Windows.

ГЛАВА 13

Поддержка сетей

Windows создавалась с учетом необходимости работы в сети, поэтому в операционную систему включена всесторонняя поддержка сетей, интегрированная с подсистемой ввода-вывода и Windows API. К четырем базовым типам сетевого программного обеспечения относятся сервисы, API, протоколы и драйверы устройств сетевых адаптеров. Все они располагаются один над другим, образуя сетевой стек. Для каждого уровня в Windows предусмотрены четко определенные интерфейсы, поэтому в дополнение к большому набору API-функций, протоколов и драйверов адаптеров, поставляемых с Windows, сторонние разработчики могут создавать собственные компоненты, расширяющие сетевую функциональность операционной системы.

В этой главе будет рассмотрен весь сетевой стек Windows — снизу доверху. Сначала мы поговорим о том, как сетевые компоненты Windows соотносятся с уровнями эталонной модели OSI (Open Systems Interconnection). Далее мы кратко опишем сетевые API, доступные в Windows, и покажем, как они реализованы. Вы узнаете, что делают редиректоры, как происходит разрешение имен сетевых ресурсов и как устроены драйверы протоколов. Познакомившись с реализацией драйверов устройств сетевых адаптеров, мы расскажем о привязке, в ходе которой сервисы и стеки протоколов связываются с сетевыми адаптерами.

Сетевая архитектура Windows

Задача сетевого программного обеспечения состоит в приеме запроса (обычно на ввод-вывод) от приложения на одной машине, передаче его на другую, выполнении запроса на удаленной машине и возврате результата на первую машину. В ходе этих операций запрос неоднократно трансформируется. Высокоуровневый запрос вроде «считать *x* байтов из файла *y* на машине *z*» требует, чтобы программное обеспечение определило, как достичь машины *z* и какой коммуникационный протокол она понимает. Затем запрос должен быть преобразован для передачи по сети — например, разбит на короткие пакеты данных. Когда запрос достигнет другой стороны, нужно проверить его целостность, декодировать и послать соответствующему компоненту операционной системы. По окончании обработки запрос должен быть закодирован для обратной передачи по сети.

Эталонная модель OSI

Чтобы помочь поставщикам в стандартизации и интеграции их сетевого программного обеспечения, международная организация по стандартизации (ISO) определила программную модель пересылки сообщений между компьютерами. Эта модель получила название *эталонной модели OSI* (Open Systems Interconnection). В ней определено семь уровней программного обеспечения (рис. 13-1).

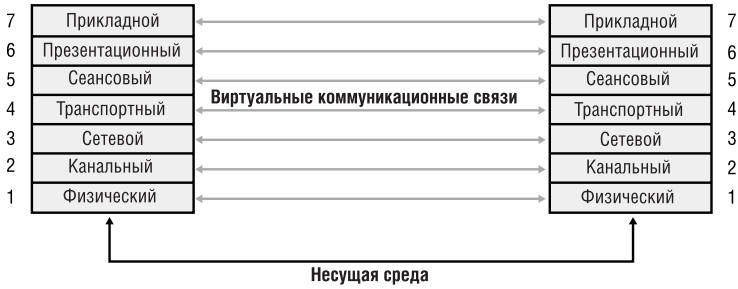


Рис. 13-1. *Эталонная модель OSI*

Эталонная модель OSI — идеал, точно реализованный лишь в очень немногих системах, но часто используемый при объяснении основных принципов работы сети. Каждый уровень на одной из машин считает, что он взаимодействует с тем же уровнем на другой машине. На данном уровне обе машины «разговаривают» на одном языке, или протоколе. Но в действительности сетевой запрос должен сначала пройти до самого нижнего уровня на первой машине, затем он передается по несущей среде и уже на второй машине вновь поднимается до уровня, который его поймет и обработает.

Задача каждого уровня в том, чтобы предоставлять сервисы более высоким уровням и скрывать от них конкретную реализацию этих сервисов. Подробное обсуждение каждого сетевого уровня выходит за рамки нашей книги, но мы все же дадим их краткое описание.

- **Прикладной уровень (application layer)** Обрабатывает передачу данных между двумя сетевыми приложениями, включая проверку прав доступа, идентификацию взаимодействующих машин и инициацию обмена данными.
- **Презентационный уровень (presentation layer)** Отвечает за форматирование данных, в том числе решает, должны ли строки заканчиваться парой символов «возврат каретки/перевод строки» (CR/LF) или только символом «возврат каретки» (CR), надо ли сжимать данные, кодировать и т. д.
- **Сеансовый уровень (session layer)** Управляет соединением взаимодействующих приложений, включая высокоуровневую синхронизацию и контроль за тем, какое из них «говорит», а какое «слушает».
- **Транспортный уровень (transport layer)** На передающей стороне разбивает сообщения на пакеты и присваивает им порядковые номера, га-

рантирующие прием пакетов в должном порядке. Кроме того, изолирует сеансовый уровень от влияния изменений в составе оборудования.

- **Сетевой уровень (network layer)** Создает заголовки пакетов, отвечает за маршрутизацию, контроль трафика и взаимодействие с межсетевой средой. Это самый высокий из уровней, который понимает топологию сетей, т. е. физическую конфигурацию машин в них, ограничения пропускной способности этих сетей и т. д.
- **Канальный уровень (data-link layer)** Пересылает низкоуровневые кадры данных, ждет подтверждений об их приеме и повторяет передачу кадров, потерянных в ненадежных линиях связи.
- **Физический уровень (physical layer)** Передает биты по сетевому кабелю или другой физической несущей среде.

Как уже говорилось, каждый сетевой уровень считает, что он взаимодействует с эквивалентным уровнем на другой машине, который использует тот же протокол. Набор протоколов, передающих запросы по сетевым уровням, называется *стеком протоколов*.

Сетевые компоненты Windows

На рис. 13-2 представлена общая схема сетевых компонентов Windows, их соответствие уровням модели OSI, а также протоколы, используемые различными уровнями. Как видите, между уровнями OSI и реальными сетевыми компонентами нет точного соответствия. Некоторые компоненты охватывают несколько уровней. Ниже приводится список сетевых компонентов с кратким описанием.

- **Сетевые API** Обеспечивают независимое от протоколов взаимодействие приложений через сеть. Сетевые API реализуются либо в режиме ядра и пользовательском режиме, либо только в пользовательском режиме. Некоторые сетевые API являются оболочками других API и реализуют специфическую модель программирования или предоставляют дополнительные сервисы. (Термином «сетевые API» обозначаются любые программные интерфейсы, предоставляемые сетевым программным обеспечением.)
- **Клиенты TDI (Transport Driver Interface)** Драйверы устройств режима ядра, обычно реализующие ту часть сетевого API, которая работает в режиме ядра. Клиенты TDI называются так из-за того, что пакеты запросов ввода-вывода (IRP), которые они посылают драйверам протоколов, форматируются по стандарту Transport Driver Interface (документированному в DDK). Этот стандарт определяет общий интерфейс программирования драйверов устройств режима ядра. (Об IRP см. главу 9.)
- **Транспорты TDI** Представляют собой драйверы протоколов режима ядра и часто называются *транспортами*, NDIS-*драйверами протоколов* или *драйверами протоколов*. Они принимают IRP от клиентов TDI и обрабатывают запросы, представленные этими IRP. Обработка запросов может потребовать взаимодействия через сеть с другим равноправным

компьютером; в таком случае транспорт TDI добавляет к данным IRP заголовки, специфичные для конкретного протокола (TCP, UDP, IPX), и взаимодействует с драйверами адаптеров через функции NDIS (также документированные в DDK). В общем, транспорты TDI связывают приложения через сеть, выполняя такие операции, как сегментация сообщений, их восстановление, упорядочение, подтверждение и повторная передача.

- **Библиотека NDIS (Ndis.sys)** Инкапсулирует функциональность для драйверов адаптеров, скрывая от них специфику среды Windows, работающей в режиме ядра. Библиотека NDIS экспортирует функции для транспортов TDI, а также функции поддержки для драйверов адаптеров.

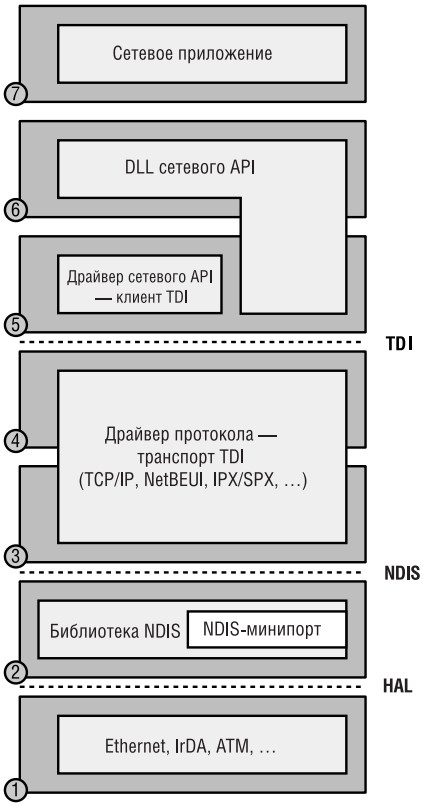


Рис. 13-2. *Модель OSI и сетевые компоненты Windows*

- **Минипорт-драйверы NDIS** Драйверы режима ядра, отвечающие за организацию интерфейсов между транспортами TDI и конкретными сетевыми адаптерами. Минипорт-драйверы NDIS пишутся так, чтобы они были заключены в оболочку библиотеки NDIS. Такая инкапсуляция обеспечивает межплатформенную совместимость с потребительскими версиями Microsoft Windows. Минипорт-драйверы NDIS не обрабатывают IRP, а регистрируют интерфейс таблицы вызовов библиотеки NDIS, которая

содержит указатели на функции, соответствующие функциям, экспортируемым библиотекой NDIS для транспортов TDI. Минипорт-драйверы NDIS взаимодействуют с сетевыми адаптерами, используя функции библиотеки NDIS, которые вызывают соответствующие функции HAL.

Фактически четыре нижних сетевых уровня часто обозначают собирательным термином «транспорт», а компоненты, расположенные на трех верхних уровнях, — термином «пользователи транспорта».

Далее мы подробно рассмотрим сетевые компоненты, показанные на рис. 13-2 (равно как и не показанные на нем), обсудим их взаимосвязи и то место, которое они занимают в Windows.

Сетевые API

Для поддержки унаследованных приложений и для совместимости с промышленными стандартами в Windows реализован целый набор сетевых API. В этом разделе мы расскажем о сетевых API и поясним, как они используются приложениями. Важно иметь в виду, что выбор API для приложения определяется характеристиками API: поверх каких протоколов он может работать, поддерживает ли он надежную и двустороннюю связь, а также переносим ли он на другие Windows-платформы, на которых может работать данное приложение. Мы обсудим следующие сетевые API:

- Windows Sockets (Winsock);
- Remote Procedure Call (RPC);
- API доступа к Web;
- именованные каналы (named pipes) и почтовые ящики (mailslots);
- NetBIOS:
- прочие сетевые API.

Windows Sockets

Изначально Windows Sockets (Winsock) версии 1.0 был Microsoft-реализацией BSD (Berkeley Software Distribution) Sockets, программного интерфейса, с 80-х годов прошлого века ставшего стандартом, на основе которого UNIX-системы взаимодействовали через Интернет. Поддержка сокетов в Windows существенно упрощает перенос сетевых приложений из UNIX в Windows. Современные версии Winsock включают большую часть функциональности BSD Sockets, а также содержат специфические расширения от Microsoft, развитие которых продолжается. Winsock поддерживает как надежные коммуникации, ориентированные на логические соединения, так и ненадежные коммуникации, не требующие логических соединений. Windows предоставляет Winsock 2.2 — для устаревших версий Windows он доступен в виде надстройки. Функциональность Winsock 2.2 выходит далеко за рамки спецификации BSD Sockets, и, в частности, он поддерживает функции, использующие средства асинхронного ввода-вывода в Windows, что обеспечивает

гораздо более высокую производительность и масштабируемость, чем исходный BSD Sockets.

Winsock обеспечивает:

- ввод-вывод по механизму «scatter/gather» и асинхронный ввод-вывод;
- поддержку Quality of Service (QoS) — если нижележащая сеть поддерживает QoS, приложения могут согласовывать между собой максимальные задержки и полосы пропускания;
- расширяемость — Winsock можно использовать не только с протоколами, которые он поддерживает в Windows, но и с другими;
- поддержку интегрированных пространств имен, отличных от определенных протоколом, который используется приложением вместе с Winsock. Например, сервер может опубликовать свое имя в Active Directory, а клиент, используя расширения пространств имен, — найти адрес сервера в Active Directory;
- поддержку многоадресных сообщений, передаваемых из одного источника сразу нескольким адресатам.

Далее мы рассмотрим принципы работы Winsock и опишем способы его расширения.

Функционирование Winsock на клиентской стороне

Первый шаг Winsock-приложения — инициализация Winsock API вызовом инициализирующей функции. В Windows 2000 такое приложение должно затем создать сокет, представляющий конечную точку коммуникационного соединения. Приложение получает адрес сервера, к которому ему нужно подключиться, вызовом *gethostbyname*. Winsock не зависит от конкретного протокола, поэтому адрес может быть указан по любому установленному в системе протоколу, поверх которого работает Winsock (TCP/IP, TCP/IP с IP версии 6, IPX). Получив адрес сервера, клиент, ориентированный на логические соединения (connection-oriented client), пытается подключиться к этому серверу, вызывая функцию *connect* и передавая ей адрес сервера.

В Windows XP и Windows Server 2003 приложение должно получить адрес сервера через *getaddrinfo*, а не *gethostbyname*. Функция *getaddrinfo* возвращает список адресов, назначенных серверу, и клиент пытается поочередно подключиться по каждому из них до тех пор, пока ему не удастся установить соединение. Это гарантирует, что клиент, поддерживающий, например, только IP версии 4 (IPv4), соединится с сервером, которому могли быть назначены как IPv4-, так и IPv6-адреса, по соответствующему IPv4-адресу.

Установив соединение, клиент может посылать и принимать данные через свой сокет, используя, например, *recv* и *send*. Клиент, не ориентированный на логические соединения (connectionless client), указывает удаленный адрес через эквивалентные функции API, не ориентированного на логические соединения; в данном случае — через *sendto* и *recvfrom* соответственно.

Функционирование Winsock на серверной стороне

Последовательность операций серверного приложения отличается от таковой для клиентского. После инициализации Winsock API сервер создает сокет и выполняет его привязку к локальному адресу через *bind*. Как и в случае клиентского приложения, тип адреса (по TCP/IP, TCP/IP с IP версии 6 или какому-то другому протоколу) выбирается серверным приложением.

Если сервер ориентирован на логические соединения, он выполняет на сокете операцию *listen*, указывая число соединений, которое он может поддерживать на этом сокете. Далее он выполняет операцию *accept*, чтобы клиент мог подключиться к сокету. При наличии ждущего запроса на соединение вызов *accept* завершается немедленно. В ином случае он завершается лишь после поступления запроса на соединение. После того как соединение установлено, функция *accept* возвращает новый сокет, представляющий серверную сторону соединения. Сервер может выполнять операции приема и передачи данных с помощью таких функций, как, например, *recv* и *send*. Рис. 13-3 иллюстрирует коммуникационную связь между клиентом и сервером Winsock, ориентированными на логические соединения.

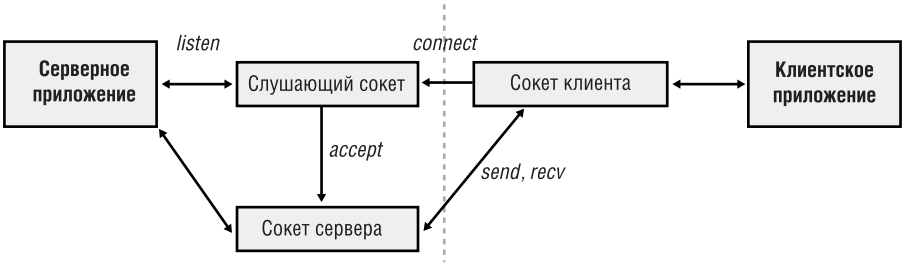


Рис. 13-3. *Так работает Winsock при использовании логических соединений*

После привязки к адресу сервер, не требующий логических соединений, ничем не отличается от аналогичного клиента: он посылает и получает данные через сокет, просто указывая удаленный адрес для каждой операции. Большинство протоколов, не ориентированных на логические соединения, ненадежны и, как правило, не позволяют определить, получил ли адресат отправленные ему пакеты данных — *дейтаграммы* (datagrams). Такие протоколы идеальны для передачи коротких сообщений, когда надежность доставки не играет определяющей роли (впрочем, приложение может само реализовать соответствующие средства поверх протокола).

Расширения Winsock

С точки зрения программирования для Windows, сильной стороной Winsock API является его интеграция с механизмом Windows-сообщений. Winsock-приложение может использовать преимущества такой интеграции для выполнения асинхронных операций с сокетом и приема уведомления о завершении операции через стандартное Windows-сообщение или функцию обратного вызова. Это упрощает разработку Windows-приложений, поскольк-

ку позволяет отказаться от многопоточности и синхронизирующих объектов для поддержки сетевого ввода-вывода и реакции на пользовательский ввод или запросы диспетчера окон на обновление окон приложения.

Кроме вспомогательных функций, прямо соответствующих функциям, реализованным в BSD Sockets, Microsoft добавила несколько функций, не входящих в стандарт Winsock. Две из них, *AcceptEx* и *TransmitFile*, стоят того, чтобы привести здесь их описание, так как благодаря им многие Web-серверы под управлением Windows достигают высокой производительности. *AcceptEx* является версией функции *accept*, которая в процессе установления соединения с клиентом возвращает адрес и первое сообщение клиента. *AcceptEx* дает возможность серверному приложению подготовиться к серии операций *accept* для последующей обработки множества входящих соединений. А это позволяет Web-серверу избежать выполнения сразу нескольких Winsock-функций.

Установив соединение с клиентом, Web-сервер обычно посылает ему файл, например Web-страницу. Реализация функции *TransmitFile* интегрирована с диспетчером кэша, что позволяет серверу посылать файл непосредственно из кэша файловой системы. Такая пересылка данных называется *нулевым копированием* (zero-copy), поскольку в этом случае серверу не приходится обращаться к файловым данным: он просто указывает описатель файла и диапазон пересылаемых байтов. Кроме того, функция *TransmitFile* позволяет серверу присоединять к началу или концу файла дополнительные данные. Это дает ему возможность посылать заголовочную информацию, например имя Web-сервера и поле, в котором указывается размер посылаемого сообщения. Internet Information Services (IIS), входящая в комплект поставки Windows, использует как *AcceptEx,* так и *TransmitFile*.

В Windows XP и Windows Server 2003 добавлен целый набор других, многофункциональных API-функций, в том числе *ConnectEx*, *DisconnectEx* и *TransmitPackets*. *ConnectEx* устанавливает соединение и посылает первое сообщение по этому соединению. *DisconnectEx* закрывает соединение и разрешает повторное использование описателя сокета, представляющего данное соединение, в вызове *AcceptEx* или *ConnectEx*. Наконец, *TransmitPackets* — полный аналог *TransmitFile* с тем исключением, что позволяет передавать не только файловые данные, но и данные, находящиеся в памяти.

Принципы расширения Winsock

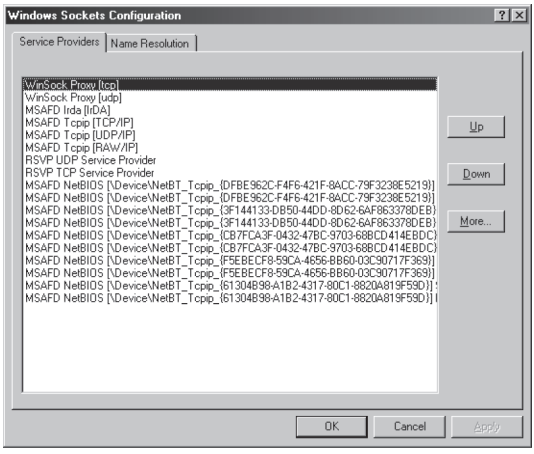
Winsock является расширяемым API, поскольку сторонние разработчики могут добавлять *провайдеры транспортных сервисов* (transport service providers, TSP), организующие интерфейсы между Winsock и другими протоколами или уровнями поверх существующих протоколов (это позволяет реализовать такую функциональность, как поддержка прокси). Сторонние разработчики также могут добавлять *провайдеры пространств имен* (namespace service providers), дополняющие механизмы разрешения имен в Winsock. Такие компоненты подключаются к Winsock через его интерфейс провайдеров сервисов (service provider interface, SPI). Если какой-то TSP регист-

рируется в Winsock, последний реализует на его основе функции сокета (вроде *connect* и *accept*) для тех типов адресов, которые указаны этим провайдером как поддерживаемые. Никаких ограничений на то, как TSP реализует функции, не налагается, но такая реализация обычно требует взаимодействия с драйвером транспорта в режиме ядра.

Требование к любому клиент-серверному приложению, использующему Winsock, заключается в следующем: сервер должен сделать свой адрес доступным клиентам, чтобы они могли подключаться к серверу. Для стандартных сервисов, выполняемых в TCP/IP, с этой целью используются так называемые общеизвестные адреса. Если браузер знает имя компьютера, на котором работает Web-сервер, он может подключиться к нему, указав общеизвестный адрес Web-сервера (к IP-адресу сервера добавляется строка «:80» — номер HTTP-порта). Провайдеры пространств имен позволяют серверам регистрировать свое присутствие и другими способами. Например, провайдер пространства имен мог бы на серверной стороне регистрировать адрес сервера в Active Directory, а на клиентской — искать его в Active Directory. Провайдеры пространств имен обеспечивают эту функциональность Winsock, реализуя такие стандартные Winsock-функции разрешения имен, как *getaddrinfo* (заменяет *gethostbyname*) и *getnameinfo*.

ЭКСПЕРИМЕНТ: просмотр провайдеров сервисов Winsock

Утилита Windows Sockets Configuration (Sporder.exe), входящая в Platform SDK, показывает зарегистрированные в Winsock провайдеры транспортных сервисов и пространств имен и позволяет изменять порядок перечисления TSP. Например, если в системе имеется два провайдера транспортных сервисов TCP/IP, то первым в списке идет TSP по умолчанию для Winsock-приложений, использующих протокол TCP/IP. На иллюстрации показано окно Sporder, в котором перечислены зарегистрированные TSP.



Реализация Winsock

Реализация Winsock представлена на рис. 13-4. Его программный интерфейс поддерживается библиотекой Ws2_32.dll (\Windows\System32\Ws2_32.dll), которая обеспечивает приложениям доступ к функциям Winsock. Для операций над именами и сообщениями Ws2_32.dll вызывает сервисы TSP и провайдеров пространств имен. Библиотека Mswsock.dll выступает в роли TSP для протоколов, поддерживаемых Microsoft в Winsock. Она взаимодействует с драйверами протоколов режима ядра с помощью вспомогательных библиотек Winsock (Winsock Helpers), специфичных для конкретного протокола. Например, Wshtcpip.dll — вспомогательная библиотека TCP/IP. В Mswsock.dll (\Windows\System32\Mswsock.dll) реализованы такие расширения Winsock, как функции *TransmitFile*, *AcceptEx* и *WSARecvEx*. Windows поставляется со вспомогательными библиотеками для TCP/IP, TCP/IP с IPv6, AppleTalk, IPX/SPX, ATM и IrDA (Infrared Data Association), а также с провайдерами пространств имен для DNS (TCP/IP), Active Directory и IPX/SPX.

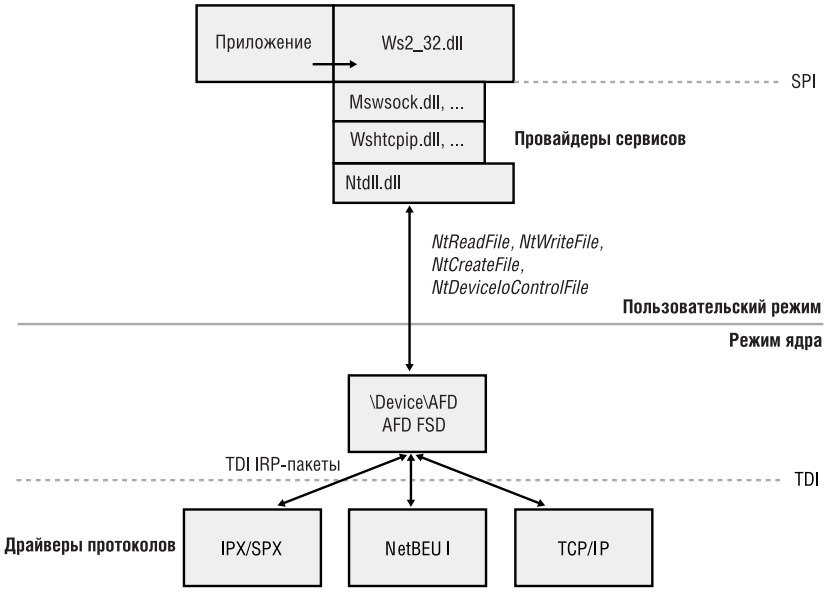


Рис. 13-4. *Реализация Winsock*

Подобно API именованных каналов и почтовых ящиков Winsock интегрируется с Windows-моделью ввода-вывода и использует для представления сокетов описатели файлов. Для этого нужна помощь со стороны драйвера файловой системы режима ядра, поэтому, реализуя функции на основе сокетов, Msafd.dll использует сервисы AFD (Ancillary Function Driver) (\Windows\System32\Drivers\Afd.sys). AFD является клиентом TDI и выполняет сетевые операции с использованием сокетов, например посылает и принимает сообщения, отправляя TDI IRP-пакеты драйверам протокола. AFD не запрограммирован на использование определенных драйверов протоколов — вместо этого

Msafd.dll уведомляет AFD о протоколе, используемом для сокета, и в результате AFD может открыть объект «устройство», представляющий этот протокол.

Windows Sockets Direct

Windows Sockets Direct (WSD) — это интерфейс, позволяющий в Winsock-приложениях без всякой модификации использовать преимущества сетей устройств хранения данных (System Area Networks, SAN). Высокопроизводительные SAN идеальны для самых разнообразных применений — от распределенных вычислений до трехуровневых архитектур электронной коммерции вроде показанной на рис. 13-5. В данной системе сети SAN соединяют Web-серверы (презентационный Web-уровень) с серверами бизнес-логики и с серверами базы данных, что обеспечивает высокоскоростную передачу данных между различными уровнями обработки информации. Поддержка WSD имеется в Windows 2003 и Windows 2000 Data Center Server, а также в Windows 2000 Advanced Server с Service Pack (SP) 2 и выше.

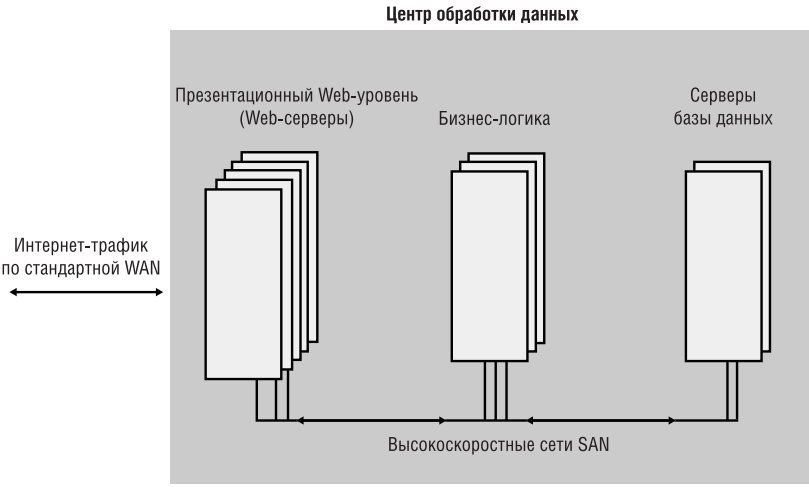


Рис. 13-5. *Применение SAN в трехуровневой архитектуре электронной коммерции*

SAN-соединения Высокая производительность SAN обычно достигается за счет специализированных сетевых соединений и коммутационного оборудования. К наиболее распространенным типам SAN-соединений относятся InfiniBand, Gigabit Ethernet, FiberChannel и различные фирменные (закрытые) решения. Физическая память, разделяемая двумя компьютерами, тоже может служить SAN-соединением.

Коммутационное оборудование SAN реализует немаршрутизируемый протокол, предоставляющий TCP-эквивалентные гарантии, в частности надежную доставку сообщений в правильном порядке. Эти аппаратные средства также поддерживают механизм SAN, называемый удаленным прямым доступом к памяти (Remote Direct Memory Access, RDMA); этот механизм

позволяет напрямую передавать сообщения из физической памяти компьютера-источника в физическую память компьютера-получателя без промежуточной операции копирования, которая обычно выполняется на стороне, принимающей сообщения. Благодаря этому RDMA освобождает процессор и шину памяти от лишней нагрузки, связанной с операцией копирования.

Реализации SAN также позволяют обходиться без обращений к компонентам режима ядра, посылая и принимая данные напрямую между пользовательскими приложениями. Это сокращает число системных вызовов, инициируемых приложениями, и соответственно уменьшает время, затрачиваемое на выполнение системного кода поддержки сетей.

Архитектура WSD Большинство реализаций SAN требуют модификации приложений для взаимодействия с сетевыми протоколами SAN и использования преимуществ аппаратно-реализованных протоколов и механизмов SAN вроде RDMA, но WSD позволяет любому Winsock-приложению, работающему по протоколу TCP, задействовать возможности SAN без такой модификации. Само название WSD подчеркивает, что он обеспечивает приложениям прямой доступ к оборудованию SAN, в обход стека TCP/IP. А сокращение пути передачи данных повышает производительность приложений в 2–2,5 раза.

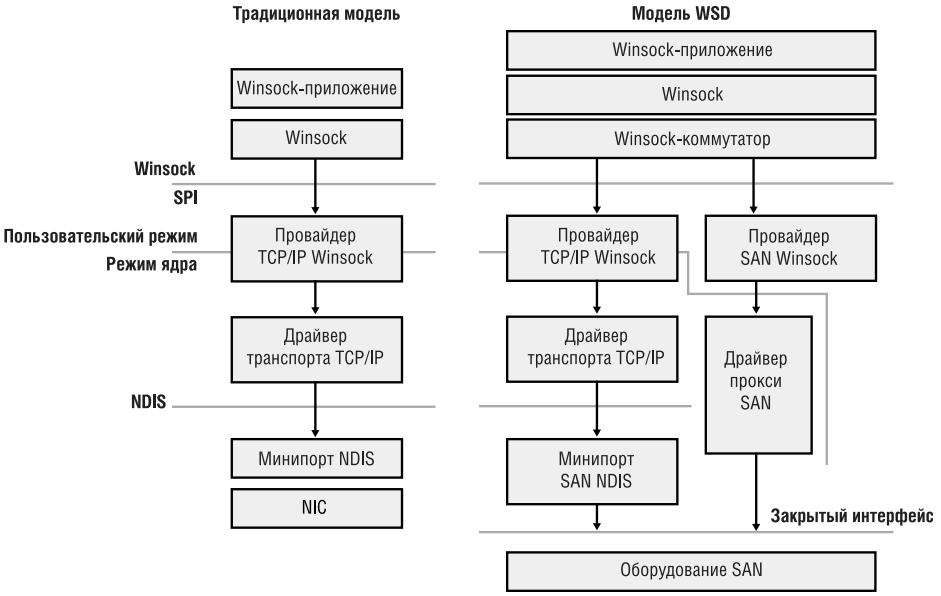


Рис. 13-6. *Традиционная модель Winsock и модель WSD*

Такое сокращение достигается за счет использования программного коммутатора, размещаемого на уровень ниже Winsock DLL, как показано на рис. 13-6. Этот коммутатор переадресует сетевые операции SAN провайдеру сервисов Winsock (Winsock service provider, WSP), который предоставляется производителем SAN. WSP служит эквивалентом NDIS-драйвера, работаю-

щим в пользовательском режиме, и может проецировать аппаратные регистры SAN на память пользовательского режима, а затем манипулировать оборудованием без участия компонентов режима ядра. Однако некоторые операции все же требуют поддержки со стороны таких компонентов, например для проецирования содержимого аппаратных регистров на память пользовательского режима; эта поддержка тоже предоставляется производителем оборудования SAN. Наконец, производитель SAN предоставляет минипорт-драйвер NDIS, выступающий в роли интерфейса между стеком TCP/IP и оборудованием SAN для приложений, которые используют сетевые средства Winsock, не поддерживаемых SAN на аппаратном уровне.

Remote Procedure Call (RPC)

RPC — стандарт сетевого программирования, разработанный в начале 80-х. Организация Open Software Foundation (теперь — The Open Group) сделала RPC частью стандарта OSF DCE (Distributed Computing Environment). Несмотря на наличие второго стандарта RPC, SunRPC, реализация RPC от Microsoft совместима со стандартом OSF DCE. RPC, опираясь на другие сетевые API (именованные каналы или Winsock), предоставляет альтернативную модель программирования, в какой-то мере скрывающую детали сетевого программирования от разработчика приложений.

Функционирование RPC

Механизм RPC позволяет создавать приложения, состоящие из произвольного числа процедур, часть которых выполняется локально, а часть — на удаленных компьютерах (через сеть). RPC предоставляет модель работы с сетью, ориентированную на процедуры, а не на транспорты, что упрощает разработку распределенных приложений.

Сетевое программное обеспечение традиционно базируется на модели обработки ввода-вывода. В Windows, например, сетевая операция начинается с того, что приложение выдает запрос на удаленный ввод-вывод. Операционная система обрабатывает запрос, передавая его редиректору, который действует в качестве удаленной файловой системы, обеспечивая прозрачное взаимодействие клиента с удаленной файловой системой. Редиректор передает запрос удаленной файловой системе, а после того как удаленная система выполнит запрос и вернет результаты, локальная сетевая плата генерирует прерывание. Ядро обрабатывает это прерывание, и исходная операция ввода-вывода завершается, возвращая результаты вызывающей программе.

RPC использует совершенно другой подход. Приложения RPC похожи на другие структурированные приложения: у них есть основная программа, которая для выполнения специфических задач вызывает процедуры или библиотеки процедур. Отличие приложений RPC от обычных программ в том, что некоторые библиотеки процедур в приложениях RPC выполняются на удаленных компьютерах, а некоторые — на локальном (рис. 13-7).

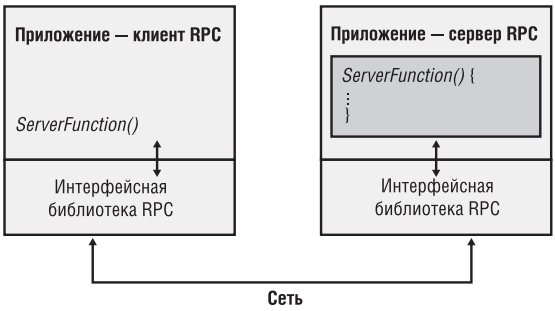


Рис. 13-7. *Так работает RPC*

Для приложения RPC все процедуры кажутся локальными. Иначе говоря, вместо того чтобы заставлять программиста писать код для передачи запросов на вычисления или ввод-вывод по сети, работы с сетевыми протоколами, обработки сетевых ошибок, ожидания результатов и т. д., программное обеспечение RPC выполняет все эти задачи автоматически. Кроме того, механизм RPC в Windows работает с любыми транспортами, которые имеются в системе.

Создавая приложение RPC, программист решает, какие процедуры будут выполняться локально, а какие — удаленно. Допустим, обычная рабочая станция подключена по сети к суперкомпьютеру Cray или к специализированной машине, предназначенной для быстрого выполнения векторных вычислений. Если программист пишет программу, работающую с большими матрицами, то с точки зрения производительности имело бы смысл переложить математические вычисления на удаленный компьютер, написав программу в виде приложения RPC.

Функционирует приложение RPC следующим образом. В процессе своей работы оно вызывает как локальные процедуры, так и процедуры, отсутствующие на локальной машине. Для обработки последнего случая приложение связывается с локальной DLL, которая содержит интерфейсные процедуры (stub procedures) для всех удаленных процедур. В простой программе интерфейсные процедуры статически связываются с приложением, но в компоненте большего размера они включаются в отдельные DLL. В DCOM обычно применяется последний метод. Интерфейсная процедура имеет то же имя и тот же интерфейс, что и удаленная процедура, но вместо выполнения соответствующей операции она просто преобразует переданные ей параметры для передачи по сети — такой процесс называется *маршалингом* (marshaling). Маршалинг заключается в упорядочении параметров и их упаковке в определенном формате.

Далее интерфейсная процедура вызывает процедуры библиотеки RPC периода выполнения, и они находят компьютер, на котором расположены удаленные процедуры, определяют используемые этим компьютером механизмы транспорта и посылают запрос при помощи локального программного обеспечения сетевого транспорта. Когда удаленный сервер получает запрос RPC, он выполняет обратное преобразование параметров (unmar-

shaling), реконструирует исходный вызов процедуры и вызывает ее. Закончив обработку, сервер выполняет обратную последовательность действий для возврата результатов вызывающей программе.

Кроме интерфейса, основанного на описанном здесь синхронном вызове процедур, RPC в Windows также поддерживает *асинхронный RPC* (asynchronous RPC). Он позволяет приложению RPC вызывать функцию и, не дожидаясь ее выполнения, продолжать свою работу. На это время приложение может перейти к выполнению другого кода. Когда от сервера придет ответ, библиотека RPC периода выполнения уведомит клиент о завершении операции. При этом используется механизм уведомления, запрошенный клиентом. Если клиент выбрал для уведомления синхронизирующий объект «событие», он ждет его перехода в свободное состояние, вызвав функцию *WaitForSingleObject* или *WaitForMultipleObject*. Если клиент предоставляет APC (Asynchronous Procedure Call), библиотека RPC периода выполнения ставит APC в очередь потока, выполняющего RPC-функцию. Если же клиент использует в качестве механизма уведомления порт завершения ввода-вывода, он должен вызвать *GetQueuedCompletionStatus*, чтобы узнать об окончании работы этой функции. Наконец, клиент может опрашивать библиотеку RPC периода выполнения о ходе выполнения операции, вызывая *RcpAsyncGetCallStatus*.

Помимо библиотеки периода выполнения в Microsoft RPC входит компилятор MIDL (Microsoft Interface Definition Language). Этот компилятор упрощает создание приложений RPC. Программист пишет набор обычных прототипов функций (предполагается, что он использует язык C или C++), описывающих удаленные процедуры, а затем помещает их в какой-либо файл. Далее он добавляет к этим прототипам нужную дополнительную информацию, например уникальный для сети идентификатор пакета процедур, номер версии и атрибуты, указывающие, являются ли параметры входными, выходными или и теми, и другими одновременно. В конечном счете программист получает файл на языке IDL (Interface Definition Language).

Подготовленный IDL-файл транслируется компилятором MIDL, который создает интерфейсные процедуры для клиентской и серверной сторон, а также заголовочные файлы, включаемые в приложение. Когда клиентское приложение связывается с файлом интерфейсных процедур, компоновщик разрешает все ссылки на удаленные процедуры. Аналогичным образом удаленные процедуры устанавливаются на серверной машине. Программист, который намерен вызывать существующее приложение RPC, должен написать только клиентскую часть программы и скомпоновать ее с локальной библиотекой RPC периода выполнения.

Библиотека RPC периода выполнения использует для взаимодействия с транспортным протоколом *универсальный интерфейс провайдеров транспорта RPC* (RPC transport provider interface). Этот интерфейс служит тонкой прослойкой между механизмом RPC и транспортом, которая увязывает операции RPC с функциями, предоставляемыми транспортом. RPC в Windows реализует DLL-модули провайдеров транспорта для именованных каналов, NetBIOS, SPX, TPC/IP и UDP. В Windows Server 2003 провайдер транс-

порта NetBIOS изъят, но добавлен провайдер для HTTP. Аналогичным образом RPC поддерживает работу с различными средствами сетевой защиты.

ПРИМЕЧАНИЕ В Windows 2000 можно написать новые DLL-модули провайдеров для поддержки дополнительных транспортов, но, начиная с Windows XP, встраивание дополнительных DLL провайдеров не поддерживается.

Большинство сетевых служб Windows является приложениями RPC, а это значит, что они могут вызываться как локальными процессами, так и процессами на удаленных машинах. Таким образом, удаленный клиентский компьютер может обращаться к службам сервера для просмотра списка общих ресурсов, открытия файлов, записи данных в очереди печати или добавления пользователей на этом сервере, либо он может вызывать Messenger Service (Службу сообщений) для посылки сообщений (конечно, при наличии соответствующих прав доступа).

Сервер может регистрировать свое имя по адресу, который будет доступен клиенту при поиске. Эта возможность, называемая *публикацией имени сервера*, реализована в RPC и интегрирована с Active Directory. Если Active Directory не установлена, служба локатора имен возвращается к широковещательной рассылке с использованием NetBIOS. Это позволяет взаимодействовать с системами под управлением Windows NT 4 и дает возможность RPC функционировать на автономных серверах и рабочих станциях.

Защита в RPC

RPC интегрирован с компонентами поддержки защиты (security support providers, SSP), что позволяет клиентам и серверам RPC использовать аутентификацию и шифрование при коммуникационной связи. Когда серверу RPC требуется защищенное соединение, он сообщает библиотеке RPC периода выполнения, какую службу аутентификации следует добавить в список доступных служб аутентификации. А когда клиенту нужно использовать защищенное соединение, он выполняет привязку к серверу. Во время привязки к серверу клиент должен указать библиотеке RPC службу аутентификации и нужный *уровень аутентификации*. Различные уровни аутентификации обеспечивают подключение к серверу только авторизованных клиентов, проверку каждого сообщения, получаемого сервером (на предмет того, послано ли оно авторизованным клиентом), контроль за целостностью RPC-сообщений и даже шифрование данных RPC-сообщений. Чем выше уровень аутентификации, тем больше требуется обработки. Клиент также может указывать *имя участника безопасности* (principal name) для сервера. *Участник безопасности* (principal) — это сущность, распознаваемая системой защиты RPC. Сервер должен зарегистрироваться в SSP под именем участника безопасности, специфичным для SSP.

SSP берет на себя все, что связано с аутентификацией и шифрованием при коммуникационной связи, не только для RPC, но и для Winsock. В Windows несколько встроенных SSP, в том числе Kerberos SSP, реализующий

аутентификацию Kerberos v5, SChannel (Secure Channel), реализующий Secure Sockets Layer (SSL), и протоколы TLS (Transport Layer Security). Если SSP не указан, программное обеспечение RPC использует встроенные средства защиты нижележащего транспорта. Одни транспорты, в частности именованные каналы и локальный RPC, имеют такие средства защиты, а другие, например TCP, — нет. В последнем случае RPC при отсутствии указанного SSP выдает небезопасные вызовы.

Еще одна функция защиты RPC позволяет серверу подменять клиент через функцию *RpcImpersonateClient*. Когда сервер заканчивает выполнение операций, потребовавших подмены клиента собой, он возвращается к использованию своих идентификационных данных защиты вызовом функции *RpcRevertToSelf* или *RpcRevertToSelfEx* (подробнее о подмене, или олицетворении, см. главу 8).

Реализация RPC

Реализация RPC изображена на рис. 13-8, где показано, что приложение на основе RPC связано с библиотекой RPC периода выполнения (\Windows\System32\Rpcrt4.dll). Последняя предоставляет для интерфейсных RPC-функций приложений функции маршалинга, а также функции для приема и передачи упакованных данных. Библиотека RPC периода выполнения включает процедуры поддержки RPC-взаимодействия через сеть и разновидность RPC под названием *локальный RPC*. Локальный RPC позволяет двум процессам взаимодействовать в одной системе, при этом библиотека RPC в качестве сетевого API использует LPC в режиме ядра (об LPC см. главу 3). Когда RPC-взаимодействие осуществляется между удаленными системами, библиотека RPC использует API-функции Winsock, именованного канала или Message Queuing.

ПРИМЕЧАНИЕ Message Queuing в Windows Server 2003 не поддерживается в качестве транспорта.

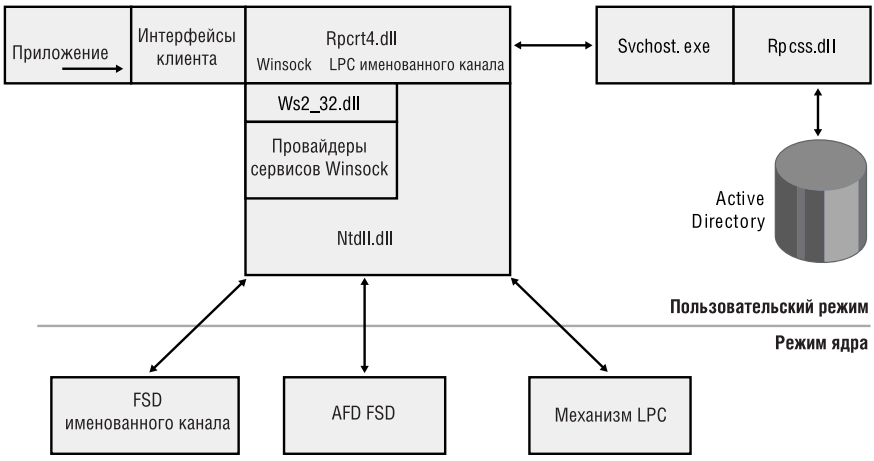


Рис. 13-8. *Реализация RPC*

Подсистема RPC (RPCSS) (\Windows\System32\Rpcss.dll) реализована в виде Windows-сервиса. RPCSS сама является приложением RPC, которое взаимодействует со своими экземплярами на других системах для поиска имен, регистрации и динамического подключения конечной точки (dynamic endpoint mapping). (Для упрощения на рис. 13-8 не показана связь RPCSS с библиотекой RPC периода выполнения.)

API-интерфейсы доступа к Web

Чтобы упростить разработку Интернет-приложений, в Windows предусмотрены клиентские и серверные API-интерфейсы доступа к Интернету. С помощью этих API приложения могут предоставлять и использовать сервисы Gopher, FTP и HTTP, не зная внутреннего устройства соответствующих протоколов. Клиентские API включают Windows Internet, также называемый WinInet (позволяет приложениям взаимодействовать с протоколами Gopher, FTP и HTTP), и WinHTTP (дает возможность приложениям взаимодействовать с протоколом HTTP). В определенных ситуациях WinHTTP удобнее WinInet. HTTP — это серверный API, введенный в Windows Server 2003 для поддержки разработки серверных Web-приложений.

WinInet

WinInet поддерживает протоколы Gopher, FTP и HTTP версий 1.0 и 1.1. Этот API делится на наборы под-API, специфичные для каждого протокола. Используя API-функции FTP вроде *InternetConnect* для подключения к FTP-серверу, *FtpFindFirstFile* и *FtpFindNextFile* для перечисления содержимого FTP-каталога, а также *FtpGetFile* и *FtpPutFile* для приема и передачи файлов, разработчик приложения может не задумываться о деталях, связанных с установлением соединения и форматированием TCP/IP-сообщений для протокола FTP. API-функции Gopher и HTTP обеспечивают аналогичный уровень абстракции. WinInet применяется базовыми компонентами Windows, например Windows Explorer и Internet Explorer.

WinHTTP

Текущая версия WinHTTP API — 5.1; она доступна в Windows 2000 с Service Pack 3, в Windows XP и Windows Server 2003. Этот API обеспечивает абстракцию протокола HTTP 1.1 для клиентских HTTP-приложений по аналогии с HTTP API в WinInet. Однако, если WinInet HTTP API предназначен для интерактивных клиентских приложений, то WinHTTP API — для серверных приложений, взаимодействующих с HTTP-серверами. Серверные приложения часто реализуются как Windows-службы без UI, поэтому им не нужны диалоговые окна, которые позволяют выводить API-функции WinInet. Кроме того, WinHTTP API лучше масштабируется и предоставляет средства защиты вроде подмены потоков, недоступные в WinInet API.

HTTP

С помощью HTTP API, реализованного в Windows Server 2003, серверные приложения могут регистрироваться на прием HTTP-запросов с определенных URL, принимать такие запросы и передавать HTTP-ответы. HTTP API включает поддержку SSL (Secure Sockets Layer), чтобы приложения могли обмениваться данными по защищенным HTTP-соединениям. Этот API поддерживает кэширование на серверной стороне, модели синхронного и асинхронного ввода-вывода, а также адресацию по IPv4 и IPv6. HTTP API используется IIS версии 6 (поставляется с Windows Server 2003).

HTTP API, к которому приложения обращаются через библиотеку Httpapi.dll, опирается на драйвер Http.sys режима ядра. Http.sys запускается по требованию при первом вызове *HttpInitialize* любым приложением. Функция *HttpCreateHttpHandle* позволяет создавать закрытую очередь запросов, а функция *HttpAddUrl* — указывать URL-адреса, по которым приложение собирается принимать запросы для обработки. Используя очереди запросов и их зарегистрированные URL, Http.sys дает возможность обслуживать HTTP-запросы на одном порту, например 80, более чем одному приложению.

HttpReceiveHttpRequest принимает входящие запросы, направленные по зарегистрированным URL, а *HttpSendHttpResponse* передает HTTP-ответы. Обе функции работают в асинхронном режиме, так что приложение может определять, закончена ли какая-то операция, используя *GetOverlappedResult* или порты завершения ввода-вывода.

Приложения могут использовать Http.sys для кэширования данных в неподкачиваемой физической памяти, вызывая *HttpAddToFragmentCache* и сопоставляя имя фрагмента с кэшируемыми данными. Для выделения неспроецированных страниц физической памяти Http.sys запускает функцию *MmAllocatePagesForMdl*, принадлежащую диспетчеру памяти. Когда Http.sys требуется сопоставление виртуального адреса с физической памятью, описываемой элементом кэша (например, если Http.sys копирует данные в кэш или передает их из кэша), он вызывает *MmMapLockedPagesSpecifyCache*, а по окончании операций — *MmUnmapLockedPages*. Http.sys хранит кэшируемые данные до тех пор, пока приложение не объявит их недействительными или пока не истечет срок их актуальности, заданный приложением. Http.sys также усекает кэшируемые данные при пробуждении рабочего потока из-за перехода в свободное состояние события, уведомляющего о малом объеме памяти (информацию об этом событии см. в главе 7). Если при вызове *HttpSendHttpResponse* приложение указывает одно или несколько имен фрагментов, Http.sys передает указатель на данные, кэшируемые в физической памяти, драйверу TCP/IP и тем самым исключает лишнюю операцию копирования.

Именованные каналы и почтовые ящики

Именованные каналы и почтовые ящики — это API, изначально разработанные Microsoft для OS/2 LAN Manager и впоследствии перенесенные в Windows NT. Именованные каналы обеспечивают надежную двустороннюю связь, тог-

да как почтовые ящики — ненадежную одностороннюю передачу данных. Преимущество почтовых ящиков — в поддержке широковещательной передачи. Оба API используют систему защиты Windows, что позволяет серверам контролировать, какие клиенты могут подключаться к ним.

Серверы назначают именованным каналами и их клиентам имена в соответствии с универсальными правилами именования (Universal Naming Convention, UNC), которые обеспечивают независимый от протоколов способ идентификации ресурсов в Windows-сетях. О реализации UNC-имен мы расскажем позже.

Функционирование именованных каналов

Коммуникационная связь по именованному каналу включает сервер именованного канала и клиент именованного канала. Сервером именованного канала является приложение, создающее именованный канал, к которому подключаются клиенты. Формат имени канала выглядит так: \\Сервер\Pipe\ИмяКанала. Элемент *Сервер* указывает компьютер, на котором работает сервер именованного канала. Элемент *Pipe* должен быть строкой «Pipe», а *ИмяКанала* — уникальное имя, назначенное именованному каналу. Уникальная часть имени канала может включать подкаталоги. Пример такого UNC-имени канала — \\MyComputer\Pipe\MyServerApp\ConnectionPipe.

Для создания именованного канала сервер использует Windows-функцию *CreateNamedPipe*. Одним из входных параметров этой функции является указатель на имя канала в форме \\.\Pipe\ИмяКанала, где «\\.\» — псевдоним локального компьютера, определенный в Windows. Функция также принимает необязательный дескриптор защиты, запрещающий несанкционированный доступ к именованному каналу, флаг, указывающий, должен ли канал быть двусторонним или односторонним, параметр, определяющий максимальное число одновременных соединений по данному каналу, и флаг режима работы канала (побайтовой передачи или передачи сообщений).

Большинство сетевых API-функций работают только в режиме побайтовой передачи. Это означает, что переданное сообщение может быть принято адресатом в виде нескольких фрагментов, из которых воссоздается полное сообщение. Именованные каналы, работающие в режиме передачи сообщений, упрощают реализацию приемника, поскольку в этом случае число передач и приемов одинаково, а приемник, разом получая целое сообщение, не должен заботиться об отслеживании фрагментов сообщений.

При первом вызове *CreateNamedPipe* с указанием какого-либо имени создается первый экземпляр именованного канала с этим именем и задается поведение всех последующих экземпляров этого канала. Повторно вызывая *CreateNamedPipe*, сервер может создавать дополнительные экземпляры именованного канала, максимальное число которых указывается при первом вызове *CreateNamedPipe*. Создав минимум один экземпляр именованного канала, сервер выполняет Windows-функцию *ConnectNamedPipe*, после чего именованный канал позволяет устанавливать соединения с клиентами. Функ-

ция *ConnectNamedPipe* может выполняться как синхронно, так и асинхронно, и она не завершится, пока клиент не установит соединение через данный экземпляр именованного канала (или не возникнет ошибка).

Для подключения к серверу клиенты именованного канала используют Windows-функцию *CreateFile* или *CallNamedPipe*, указывая при вызове имя созданного сервером канала. Если сервер вызывает функцию *ConnectNamedPipe*, профиль защиты клиента и запрошенные им права доступа к каналу (для чтения или записи) сравниваются с дескриптором защиты канала (подробнее об алгоритмах проверки прав доступа см. главу 8). Если клиенту разрешен доступ к именованному каналу, он получает описатель, представляющий клиентскую сторону именованного канала, и функция *ConnectNamedPipe*, вызванная сервером, завершается.

После того как соединение по именованному каналу установлено, клиент и сервер могут использовать его для чтения и записи данных через Windows-функции *ReadFile* и *WriteFile*. Именованные каналы поддерживают как синхронную, так и асинхронную передачу сообщений. Взаимодействие клиента и сервера через именованный канал показано на рис. 13-9.

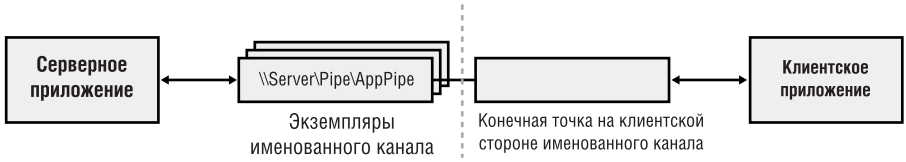


Рис. 13-9. *Связь через именованный канал*

Уникальная особенность API именованного канала заключается в том, что он позволяет серверу олицетворять клиент с помощью функции *ImpersonateNamedPipeClient*. О том, как используется олицетворение в клиент-серверных приложениях, см. раздел «Олицетворение» главы 8.

Функционирование почтового ящика

Почтовые ящики предоставляют механизм ненадежного одностороннего широковещания. Одним из примеров приложений, использующих этот тип коммуникационной связи, является сервис синхронизации времени, который каждые несколько секунд широковещательно рассылает в пределах домена сообщение с эталонным временем. Такие сообщения не критичны для работы компьютеров в сети, поэтому они рассылаются через почтовые ящики.

Как и именованные каналы, почтовые ящики интегрированы с Windows API. Сервер почтового ящика создает почтовый ящик вызовом *CreateMailslot*. Входным параметром этой функции является имя в форме «\\.\Mailslot\ИмяПочтовогоЯщика». Сервер может создавать почтовые ящики только на той машине, на которой он работает, а назначаемые им имена почтовых ящиков могут включать подкаталоги. *CreateMailslot* также принимает необязательный дескриптор защиты, контролирующий доступ клиента к почтовому ящику. Описатели, возвращаемые *CreateMailslot*, являются *перекрытыми*.

Это означает, что операции с использованием таких описателей (например, рассылка и получение сообщений) выполняются асинхронно.

Поскольку почтовые ящики поддерживают одностороннюю ненадежную передачу, число параметров *CreateMailslot* меньше, чем у *CreateNamedPipe*. После создания почтового ящика сервер просто отслеживает поступающие клиентские сообщения, вызывая функцию *ReadFile* и указывая описатель, представляющий почтовый ящик.

Клиенты почтового ящика используют формат именования, аналогичный применяемому клиентами именованных каналов, за исключением вариаций, необходимых для широковещательной передачи сообщений всем почтовым ящикам с данным именем в домене клиента или в другом указанном домене. Чтобы послать сообщение в определенный экземпляр почтового ящика, клиент вызывает функцию *CreateFile*, указывая имя, специфичное для компьютера, например «\\Сервер\Mailslot\ИмяПочтовогоЯщика». (Для представления локального компьютера клиент задает «\\.\».) Если клиент хочет получить описатель, представляющий все почтовые ящики с заданным именем в домене, членом которого он является, он указывает имя в формате «*\Mailslot\ИмяПочтовогоЯщика». Для широковещательной передачи во все почтовые ящики с заданным именем в другом домене используется имя в формате «\\ИмяДомена\Mailslot\ИмяПочтовогоЯщика».

Получив описатель, представляющий клиентскую сторону почтового ящика, клиент посылает сообщения через функцию *WriteFile*. Реализация почтовых ящиков допускает широковещательную передачу сообщений длиной не более 425 байтов. Если длина сообщения превышает 425 байтов, почтовый ящик использует механизм надежной коммуникационной связи, требующий соединения клиента с сервером по типу «один к одному», что исключает возможность широковещательной передачи. Другая (довольно странная) особенность почтовых ящиков — урезание сообщений с исходной длиной в 425 или 426 байтов до 424 байтов. Таким образом, почтовые ящики непригодны для рассылки сообщений, длина которых превышает 424 байта. На рис. 13-10 показан пример широковещательной передачи клиентского сообщения на несколько серверов почтовых ящиков в пределах домена.

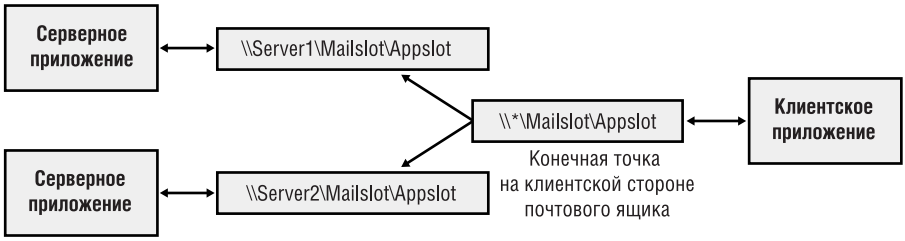


Рис. 13-10. *Широковещательная передача с помощью почтовых ящиков*

Реализация именованных каналов и почтовых ящиков

О тесной интеграции функций именованных каналов и почтовых ящиков с Windows свидетельствует тот факт, что все они реализованы в Kernel32.dll. *ReadFile* и *WriteFile*, используемые приложениями для обмена сообщениями через именованные каналы и почтовые ящики, являются основными Windows-функциями ввода-вывода. *CreateFile*, с помощью которой клиент открывает именованный канал или почтовый ящик, также является стандартной Windows-функцией ввода-вывода. Однако имена, указываемые приложениями при использовании именованных каналов и почтовых ящиков, относятся к пространству имен под управлением драйверов файловых систем именованных каналов (\Windows\System32\Drivers\Npfs.sys) и почтовых ящиков (\Windows\System32\Drivers\Msfs.sys), как показано на рис. 13-11. Драйвер файловой системы именованных каналов создает объект «устройство» \Device\NamedPipe и символьную ссылку на этот объект с именем \Global??\Pipe (\??\Pipe в Windows 2000), а драйвер файловой системы почтовых ящиков создает объект «устройство» \Device\Mailslot и символьную ссылку \Global??\Mailslot (\??\Mailslot в Windows 2000), которая указывает на этот объект. (О каталоге \Global?? диспетчера объектов см. главу 3.) Префикс «\\.\» в именах «\\.\Pipe\...» и «\\.\Mailslot\...», передаваемых *CreateFile*, транслируется в «\Global??\», чтобы эти имена разрешались через символьную ссылку на объект «устройство». Специальные функции *CreateNamedPipe* и *CreateMailslot* используют соответствующие функции ядра *NtCreateNamedPipeFile* и *NtCreateMailslotFile*.

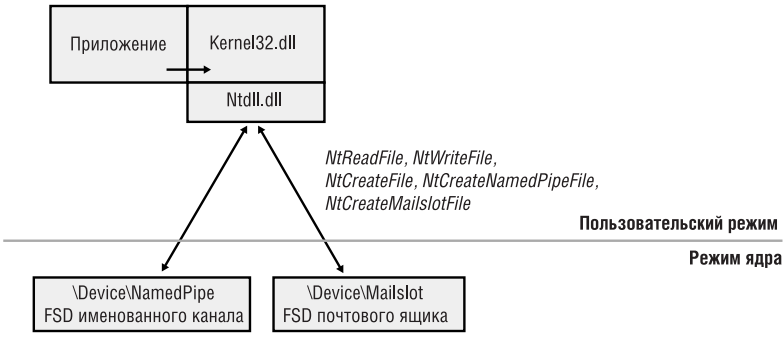


Рис. 13-11. *Реализация именованного канала и почтового ящика*

Позже мы обсудим, как драйвер файловой системы участвует в поиске удаленной системы по имени, которое задает удаленный именованный канал или почтовый ящик. Однако, когда именованный канал либо почтовый ящик создается сервером или открывается клиентом, в конечном счете вызывается соответствующий драйвер файловой системы (FSD) на той машине, где находится именованный канал или почтовый ящик. Именованные каналы и почтовые ящики реализованы в виде FSD режима ядра по нескольким причинам. Основной из них является интеграция с пространством имен

диспетчера объектов, что позволяет использовать объекты «файл» для представления открытых именованных каналов и почтовых ящиков. Подобная интеграция дает следующие преимущества.

- Используя функции защиты режима ядра, FSD реализуют для именованных каналов и почтовых ящиков стандартную защиту Windows.
- Поскольку FSD интегрированы с пространством имен диспетчера объектов, приложения могут открывать именованный канал или почтовый ящик вызовом функции *CreateFile*.
- Приложения могут взаимодействовать с именованными каналами и почтовыми ящиками через Windows-функции вроде *ReadFile* и *WriteFile*.
- FSD полагаются на диспетчер объектов в поддержке счетчиков описателей и ссылок для объектов «файл», представляющих именованные каналы и почтовые ящики.
- FSD могут реализовать собственные пространства имен каналов и почтовых ящиков, допускающие указание подкаталогов.

Так как взаимодействие через сеть при разрешении имен именованных каналов и почтовых ящиков осуществляется через редиректор, FSD при этом неявно используют протокол CIFS (Common Internet File System). Поскольку CIFS способен работать с TCP/IP, TCP/IP с IPv6 и IPX, именованные каналы и почтовые ящики доступны приложениям, выполняемым в системах, где установлен хотя бы один общий такой протокол. (Сведения о CIFS см. в главе 12.)

ЭКСПЕРИМЕНТ: просмотр пространства имен именованных каналов и наблюдение за активностью таких каналов

Открыть корневой каталог FSD именованных каналов и перечислить его содержимое с помощью Windows API нельзя — для этого нужно воспользоваться сервисами встроенного API. Утилита PipeList (*www.sysinternals.com*) перечисляет именованные каналы, определенные на компьютере, число созданных экземпляров канала с данным именем и максимальное число каналов, заданное сервером при вызове *CreateNamedPipe*. Вот пример вывода PipeList.

```
C:\>pipelist

PipeList v1.01
by Mark Russinovich
http://www.sysinternals.com

Pipe Name                          Instances       Max Instances
----------                         ---------       -------------
TerminalServer\AutoReconnect           1               1
InitShutdown                           2              -1
lsass                                  4              -1
protected_storage                      2              -1
ntsvcs                                58              -1
```

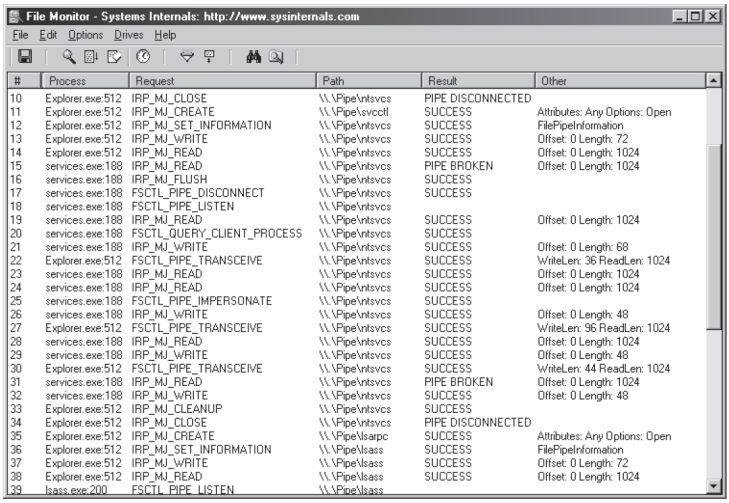
```
scerpc                                  2              -1
net\NtControlPipe1                      1               1
net\NtControlPipe2                      1               1
ExtEventPipe_Service                    1              30
net\NtControlPipe3                      1               1
Winsock2\CatalogChangeListener-37c-0    1               1
net\NtControlPipe4                      1               1
net\NtControlPipe0                      1               1
DhcpClient                              1              -1
ProfMapApi                              2              -1
winlogonrpc                             2              -1
net\NtControlPipe5                      1               1
SfcApi                                  2              -1
net\NtControlPipe6                      1               1
Winsock2\CatalogChangeListener-4c4-0    1               1
atsvc                                   2              -1
epmapper                                2              -1
net\NtControlPipe7                      1               1
spoolss                                 2              -1
wkssvc                                  3              -1
DAV RPC SERVICE                         2              -1
net\NtControlPipe8                      1               1
keysvc                                  2              -1
net\NtControlPipe9                      1               1
PCHHangRepExecPipe                      1               8
PCHFaultRepExecPipe                     1               8
net\NtControlPipe10                     1               1
OOOOOOac.OOO                            2              -1
Winsock2\CatalogChangeListener-ac-0     1               1
net\NtControlPipe11                     1               1
net\NtControlPipe12                     1               1
winreg                                  2              -1
SECLOGON                                2              -1
srvsvc                                  3              -1
ipsec                                   2              -1
trkwks                                  2              -1
Winsock2\CatalogChangeListener-ac-1     1               1
net\NtControlPipe13                     1               1
vmware-authdpipe                        1              -1
W32TIME                                 2              -1
net\NtControlPipe14                     1               1
net\NtControlPipe15                     1               1
Winsock2\CatalogChangeListener-e8-0     1               1
INETINFO                                2              -1
SMTPSVC                                 2              -1
browser                                 3              -1
net\NtControlPipe16                     1               1
Ctx_WinStation_API_service              2              -1
```

см. след. стр.

```
ExtEventPipe_s0                          1              30
ExtEventPipe_s0_poller                   1              30
net\NtControlPipe17                      1               1
interbas\server\gds_db                   1              -1
ROUTER                                   8              -1
PIPE_EVENTROOT\CIMV2SCM EVENT PROVIDER   2              -1
net\NtControlPipe20                      1               1
```

Из этого листинга ясно, что некоторые системные компоненты используют именованные каналы как механизм связи. Например, канал *InitShutdown* создан Winlogon для приема удаленных команд на завершение работы, а канал *SecLogon* — сервисом SecLogon для выполнения операций входа в интересах утилиты Runas. Определить, каким из процессов открыт каждый из этих каналов, можно с помощью утилиты Process Explorer (*www.sysinternals.com*). Заметьте, что значение Max Instances, равное –1, означает, что на число экземпляров канала с данным именем не накладывается никаких ограничений.

Драйвер фильтра файловой системы Filemon (*www.sysinternals.com*) способен подключаться к драйверу файловой системы Npfs.sys или Msfs.sys, что позволяет ему наблюдать за активностью всех именованных каналов или почтовых ящиков в системе. Для подключения Filemon к соответствующему драйверу выберите из меню Drives команду Named Pipes или Mail Slots. На иллюстрации ниже показано окно Filemon, в котором сообщается об активности именованных каналов, вызываемой двойным щелчком значка My Network Places (Мое сетевое окружение) на рабочем столе. Заметьте, что сообщения передаются через именованные каналы LSASS и службы рабочей станции.



#	Process	Request	Path	Result	Other
10	Explorer.exe:512	IRP_MJ_CLOSE	\\.\Pipe\ntsvcs	PIPE DISCONNECTED	
11	Explorer.exe:512	IRP_MJ_CREATE	\\.\Pipe\svcctl	SUCCESS	Attributes: Any Options: Open
12	Explorer.exe:512	IRP_MJ_SET_INFORMATION	\\.\Pipe\ntsvcs	SUCCESS	FilePipeInformation
13	Explorer.exe:512	IRP_MJ_WRITE	\\.\Pipe\ntsvcs	SUCCESS	Offset: 0 Length: 72
14	Explorer.exe:512	IRP_MJ_READ	\\.\Pipe\ntsvcs	SUCCESS	Offset: 0 Length: 1024
15	services.exe:188	IRP_MJ_READ	\\.\Pipe\ntsvcs	PIPE BROKEN	Offset: 0 Length: 1024
16	services.exe:188	IRP_MJ_FLUSH	\\.\Pipe\ntsvcs	SUCCESS	
17	services.exe:188	FSCTL_PIPE_DISCONNECT	\\.\Pipe\ntsvcs	SUCCESS	
18	services.exe:188	FSCTL_PIPE_LISTEN	\\.\Pipe\ntsvcs		
19	services.exe:188	IRP_MJ_READ	\\.\Pipe\ntsvcs	SUCCESS	Offset: 0 Length: 1024
20	services.exe:188	FSCTL_QUERY_CLIENT_PROCESS	\\.\Pipe\ntsvcs	SUCCESS	
21	services.exe:188	IRP_MJ_WRITE	\\.\Pipe\ntsvcs	SUCCESS	Offset: 0 Length: 68
22	Explorer.exe:512	FSCTL_PIPE_TRANSCEIVE	\\.\Pipe\ntsvcs	SUCCESS	WriteLen: 36 ReadLen: 1024
23	services.exe:188	IRP_MJ_READ	\\.\Pipe\ntsvcs	SUCCESS	Offset: 0 Length: 1024
24	services.exe:188	IRP_MJ_READ	\\.\Pipe\ntsvcs	SUCCESS	Offset: 0 Length: 1024
25	services.exe:188	FSCTL_PIPE_IMPERSONATE	\\.\Pipe\ntsvcs	SUCCESS	
26	services.exe:188	IRP_MJ_WRITE	\\.\Pipe\ntsvcs	SUCCESS	Offset: 0 Length: 48
27	Explorer.exe:512	FSCTL_PIPE_TRANSCEIVE	\\.\Pipe\ntsvcs	SUCCESS	WriteLen: 96 ReadLen: 1024
28	services.exe:188	IRP_MJ_READ	\\.\Pipe\ntsvcs	SUCCESS	Offset: 0 Length: 1024
29	services.exe:188	IRP_MJ_WRITE	\\.\Pipe\ntsvcs	SUCCESS	Offset: 0 Length: 48
30	Explorer.exe:512	FSCTL_PIPE_TRANSCEIVE	\\.\Pipe\ntsvcs	SUCCESS	WriteLen: 44 ReadLen: 1024
31	services.exe:188	IRP_MJ_READ	\\.\Pipe\ntsvcs	PIPE BROKEN	Offset: 0 Length: 1024
32	services.exe:188	IRP_MJ_WRITE	\\.\Pipe\ntsvcs	SUCCESS	Offset: 0 Length: 48
33	Explorer.exe:512	IRP_MJ_CLEANUP	\\.\Pipe\ntsvcs	SUCCESS	
34	Explorer.exe:512	IRP_MJ_CLOSE	\\.\Pipe\ntsvcs	PIPE DISCONNECTED	
35	Explorer.exe:512	IRP_MJ_CREATE	\\.\Pipe\lsarpc	SUCCESS	Attributes: Any Options: Open
36	Explorer.exe:512	IRP_MJ_SET_INFORMATION	\\.\Pipe\lsass	SUCCESS	FilePipeInformation
37	Explorer.exe:512	IRP_MJ_WRITE	\\.\Pipe\lsass	SUCCESS	Offset: 0 Length: 72
38	Explorer.exe:512	IRP_MJ_READ	\\.\Pipe\lsass	SUCCESS	Offset: 0 Length: 1024
39	lsass.exe:200	FSCTL_PIPE_LISTEN	\\.\Pipe\lsass		

NetBIOS

До начала 90-х годов NetBIOS (Network Basic Input/Output System) API был самым популярным интерфейсом программирования для персональных компьютеров. NetBIOS поддерживал связь как надежную, ориентированную на логические соединения, так и ненадежную, не требующую логических соединений. Windows поддерживает NetBIOS для совместимости с унаследованными приложениями. Microsoft не рекомендует разработчикам приложений использовать NetBIOS, поскольку существуют куда более гибкие и переносимые API, например именованные каналы и Winsock. NetBIOS в Windows поддерживается протоколами TCP/IP и IPX/SPX.

NetBIOS-имена

NetBIOS использует правила именования, согласно которым компьютерам и сетевым службам назначаются 16-байтовые имена, называемые NetBIOS-именами; 16-й байт в NetBIOS-имени интерпретируется как модификатор, который указывает, является ли имя уникальным или групповым.* Уникальное NetBIOS-имя может быть назначено только одному компьютеру или службе в сети, а групповое имя — нескольким компьютерам или службам. Адресуя сообщение на групповое имя, клиент может вести широковещательную рассылку.

Windows — для поддержки взаимодействия с системами под управлением Windows NT 4 и потребительских версий Windows — автоматически определяет NetBIOS-имя для домена как первые 15 байтов DNS-имени (Domain Name System), назначенного домену администратором. Например, домен *mspress.microsoft.com* получает NetBIOS-имя *mspress*. Аналогичным образом Windows требует, чтобы во время установки администратор назначил каждому компьютеру NetBIOS-имя.

Еще одна концепция, используемая в NetBIOS, — номера адаптеров LAN (LANA). Номер LANA присваивается каждому NetBIOS-совместимому протоколу, расположенному на более высоком уровне, чем сетевой адаптер. Так, если в компьютере установлено два сетевых адаптера, доступных для TCP/IP и NWLink, то в результате будет назначено четыре номера LANA. Номера LANA важны, поскольку приложения NetBIOS должны явно закреплять имена своих сервисов за каждым LANA, через который они готовы принимать клиентские соединения. Если приложение ждет соединений с клиентами по определенному имени, клиенты получат доступ к приложению только через протоколы, для которых зарегистрировано это имя.

Разрешение NetBIOS-имен в IP-адреса описывается в разделе «Windows Internet Name Service» далее в этой главе.

---

* Здесь авторы допускают неточность. 16-й байт NetBIOS-имени прежде всего является идентификатором типа ресурса. Он указывает сетевой компонент или службу, которая назначила это NetBIOS-имя компьютеру, пользователю или домену. Ну и, кроме того, NetBIOS-имя может быть зарегистрировано как уникальное (принадлежащее одному владельцу) или как групповое (принадлежащее нескольким владельцам). — *Прим. перев.*

Функционирование NetBIOS

Серверное приложение NetBIOS использует NetBIOS API для перечисления LANA, имеющихся в системе, и назначения каждому из них NetBIOS-имени, представляющего сервис приложения. Если сервер требует логических соединений, он выполняет NetBIOS-команду *listen* для ожидания попыток подключения клиентов. После того как соединение с клиентом установлено, сервер выполняет функции NetBIOS для передачи и приема данных. Аналогичным образом осуществляется и связь, не требующая логических соединений, но сервер просто принимает сообщения, не устанавливая соединение.

Клиент, ориентированный на логические соединения, устанавливает соединение с сервером NetBIOS, а затем с помощью функций NetBIOS передает и принимает данные. Установленное NetBIOS-соединение также называется *сеансом* (session). Если клиент хочет посылать сообщения без установления логического соединения, он просто указывает NetBIOS-имя сервера при вызове функции передачи данных.

NetBIOS состоит из набора функций, но все они действуют через один и тот же интерфейс — *Netbios*. Это наследие тех времен, когда NetBIOS реализовали в виде прерывания MS-DOS. Приложение NetBIOS выполняло прерывание MS-DOS и передавало NetBIOS структуру данных, где задавались все параметры нужной команды. В итоге функция *Netbios* в Windows принимает единственный параметр, который представляет собой структуру данных с параметрами, специфичными для запрошенного приложением сервиса.

ЭКСПЕРИМЕНТ: просмотр NetBIOS-имен через Nbtstat

Для вывода списка активных сеансов в системе, кэшируемых сопоставлений NetBIOS-имен и IP-адресов, а также NetBIOS-имен, определенных на компьютере, можно использовать встроенную в Windows команду *Nbtstat*. Ниже приведен пример вывода этой команды с параметром *–n*, при указании которого выводится список NetBIOS-имен, определенных на компьютере.

```
C:\>nbtstat -n
Local Area Connection:
Node IpAddress: [10.1.23.147] Scope Id: []
             NetBIOS Local Name Table
      Name               Type         Status
   ---------------------------------------------
    MARKLAP        <00>  UNIQUE      Registered
    WORKGROUP      <00>  GROUP       Registered
    MARKLAP        <20>  UNIQUE      Registered
    WORKGROUP      <00>  GROUP       Registered
    WORKGROUP      <1E>  GROUP       Registered
    WORKGROUP      <1D>  UNIQUE      Registered
    ..__MSBROWSE__.<01>  GROUP       Registered
```

```
Wireless Network Connection:
Node IpAddress: [0.0.0.0] Scope Id: []

No names in cache
```

Реализация NetBIOS API

Компоненты, реализующие NetBIOS API, показаны на рис. 13-12. Функция *Netbios* экспортируется приложениям из \Windows\System32\Netapi32.dll. Netapi32.dll открывает описатель драйвера режима ядра под названием *эмулятор NetBIOS* (\Windows\System32\Drivers\Netbios.sys) и выдает Windows-команды *DeviceIoControlFile* от имени приложения. Эмулятор NetBIOS транслирует команды NetBIOS в команды TDI, посылаемые драйверам протоколов.

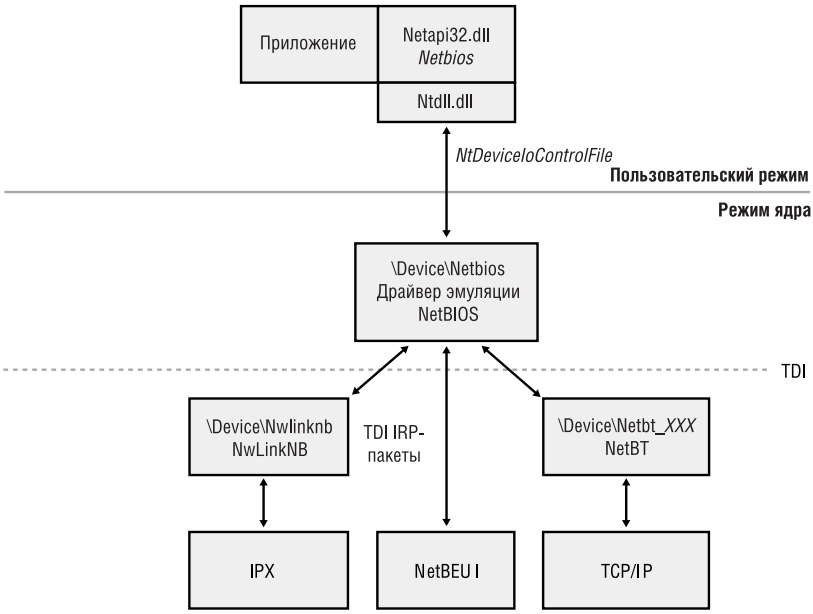


Рис. 13-12. *Реализация NetBIOS API*

Если приложение требует использовать NetBIOS поверх TCP/IP, то эмулятору NetBIOS нужен драйвер NetBT (\Windows\System32\Drivers\Netbt.sys). NetBT отвечает за поддержку семантики NetBIOS, присущей не TCP/IP, а NetBEUI (NetBIOS Extended User Interface), который включался в предыдущие версии Windows. Например, NetBIOS полагается на NetBEUI-поддержку режима передачи данных в виде сообщений и на средства разрешения имен, поэтому драйвер NetBT реализует их поверх протокола TCP/IP. Аналогичным образом драйвер NwLinkNB реализует семантику NetBIOS поверх IPX/SPX.

Другие сетевые API

В Windows входит еще несколько сетевых API, которые используются реже и расположены на более высоком уровне, чем уже рассмотренные API. Изучение всех этих API выходит за рамки книги, но четыре из них — RTC (Real-Time Communications), DCOM (Distributed Component Object Model), Message Queuing и UPnP (Universal Plug and Play) — достаточно важны для функционирования Windows и многих приложений и поэтому заслуживают краткого описания.

RTC

RTC Client API, доступный в Windows XP и Windows Server 2003, позволяет разработчикам создавать приложения, способные устанавливать многорежимные коммуникационные соединения и превращать персональный компьютер (ПК) в центр домашних или деловых коммуникаций. Голосовая и видеосвязь, мгновенный обмен сообщениями (Instant Messaging, IM), поддержка совместной работы — все это становится доступным в одном сеансе коммуникационной связи. Помимо сеансов связи между ПК, этот API позволяет устанавливать сеансы связи «ПК-телефон», «телефон-телефон» или только текстового IM. В сеансах связи между ПК также доступны совместное использование приложений (application sharing) и общая электронная доска (whiteboard).

RTC поддерживает информацию о присутствии (presence information), на основе которой клиенты могут связываться с контактами через сервер-регистратор (registrar server), хранящий информацию о текущих адресах контактов. Адресом контакта может быть ПК или телефон, а в будущем и мобильный телефон, пейджер или другое карманное устройство. Например, если приложение пытается связаться с контактом по его рабочему адресу и информация о присутствии указывает на то, что данный контакт доступен через домашний ПК, RTC автоматически перенаправит соединение на этот адрес. RTC API также обеспечивает невмешательство в частную жизнь, позволяя блокировать определенные вызовы.

DCOM

Microsoft COM API позволяет составлять приложения из компонентов, и каждый компонент представляет собой заменяемый самодостаточный модуль. Любой COM-объект экспортирует объектно-ориентированный интерфейс для манипулирования своими данными. Поскольку COM-объекты предоставляют четко определенные интерфейсы, разработчики могут реализовать новые объекты для расширения существующих интерфейсов и динамического добавления новой функциональности в приложения.

DCOM — это расширение COM, которое дает возможность размещать компоненты приложения на разных компьютерах, при этом приложению безразлично, что один COM-объект находится на локальном компьютере, а другой — на каком-то компьютере в локальной сети. Таким образом, DCOM упрощает разработку распределенных приложений. DCOM не является автономным API — в своей работе он опирается на RPC.

Message Queuing

Message Queuing представляет собой универсальную платформу для разработки распределенных приложений, использующих преимущества свободно связанного обмена сообщениями (loosely coupled messaging). Поэтому Message Queuing является также API-интерфейсом и инфраструктурой передачи сообщений. Гибкость Message Queuing определяется тем, что его очереди служат репозитариями сообщений, в которые отправители помещают сообщения для посылки получателям и из которых получатели извлекают адресованные им сообщения. Отправителям и получателям не требуется ни устанавливать соединения, ни работать в одно и то же время. А это позволяет асинхронно обмениваться сообщениями, не устанавливая прямых соединений.

Примечательная особенность Message Queuing — его интеграция с MTS (Microsoft Transaction Server) и SQL Server, что дает возможность Message Queuing участвовать в транзакциях, координируемых MS DTC (Microsoft Distributed Transaction Coordinator). Используя MS DTC с Message Queuing, можно разрабатывать для трехуровневых приложений надежные компоненты, отвечающие за обработку транзакций.

UPnP

Universal Plug and Play (UPnP) — это распределенная, открытая сетевая архитектура для поддержки соединений с интеллектуальными устройствами и точками управления (control points), подключенными к домашним сетям, интрасетям или напрямую к Интернету. Она построена на принятых стандартах и опирается на существующие технологии TCP/IP и Web. UPnP не требует конфигурирования и поддерживает автоматическое распознавание широкого спектра устройств. Она позволяет устройству динамически подключаться к сети, получать IP-адрес и сообщать о своих возможностях, когда поступает соответствующий запрос. Точки управления могут применять Control Point API в сочетании с технологией UPnP для определения присутствия и возможностей остальных устройств в сети. Устройство может автоматически выходить из сети, если оно больше не используется.

Поддержка нескольких редиректоров

У приложений есть два способа просмотра удаленных ресурсов или доступа к ним. Один из них заключается в использовании стандарта UNC и прямой адресации к удаленным ресурсам через Windows-функции, а второй — в применении Windows Networking (WNet) API для перечисления компьютеров и экспортируемых ими ресурсов. Оба подхода опираются на возможности редиректора. Для доступа клиентов к CIFS-серверам Microsoft поставляет редиректор CIFS, у которого есть компонент режима ядра (FSD редиректора) и компонент пользовательского режима (служба рабочей станции). Microsoft также предоставляет редиректор, способный обращаться к ресурсам серверов Novell NetWare, а сторонние разработчики могут добавлять в

Windows собственные редиректоры. В этом разделе мы расскажем о программном обеспечении, которое решает, какой редиректор следует вызвать для обработки запроса на удаленный ввод-вывод. За это отвечают следующие компоненты.

- **Маршрутизатор многосетевого доступа (multiple provider router, MPR)** Это DLL (\Windows\System32\Mpr.dll), определяющая, к какой сети следует обратиться, когда приложение использует Windows WNet API для просмотра удаленной файловой системы.
- **Многосетевой UNC-провайдер (multiple UNC provider, MUP)** Драйвер (\Windows\System32\Drivers\Mup.sys), определяющий, к какой сети следует обратиться, когда приложение использует Windows API ввода-вывода для открытия удаленных файлов.

Маршрутизатор многосетевого доступа

Windows-функции WNet позволяют приложениям (включая Windows Explorer и My Network Places) подключаться к сетевым ресурсам (файлам и принтерам), а также просматривать содержимое удаленных файловых систем любого типа. Так как этот API предназначен для работы с различными сетями и по разным протоколам, необходимо специальное программное обеспечение, способное посылать запросы по сети и правильно интерпретировать результаты, получаемые от удаленных серверов. Это программное обеспечение показано на рис. 13-13.

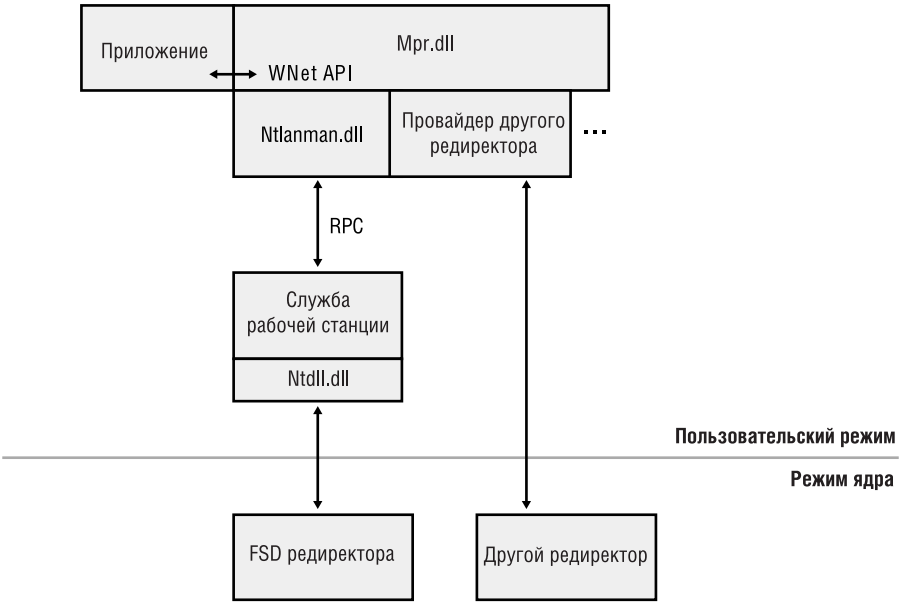


Рис. 13-13. *Компоненты MPR*

Провайдер (provider) — это программный компонент, позволяющий Windows выступать в качестве клиента какого-либо удаленного сервера. В число операций, выполняемых провайдером WNet, входят установление и разрыв сетевых соединений, удаленная печать и передача данных. Встроенный провайдер WNet включает DLL, службу рабочей станции и редиректор. Поставщики других сетей должны предоставлять только DLL и редиректор.

Когда приложение вызывает некую функцию WNet, этот вызов передается непосредственно MPR DLL. MPR принимает вызов и определяет, какой из провайдеров WNet распознает запрошенный ресурс. Все DLL провайдеров, расположенные ниже MPR, предоставляют набор стандартных функций, в совокупности называемых *интерфейсом сетевого доступа* (network provider interface). Этот интерфейс позволяет MPR определить, к какой сети пытается обратиться приложение, и направить вызов соответствующему провайдеру WNet. Провайдером службы рабочей станции является \Windows\System32\Ntlanman.dll, что указывается в параметре ProviderPath в разделе реестра HKLM\SYSTEM\CurrentControlSet\Services\LanManWorkstation\NetworkProvider.

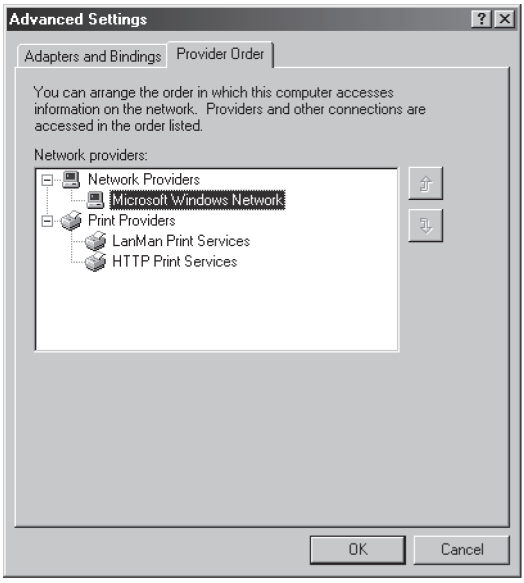


Рис. 13-14. *Редактор порядка провайдеров (служб доступа к сети)*

Когда MPR вызывается для подключения к удаленному сетевому ресурсу API-функцией *WNetAddConnection*, он просматривает в реестре параметр HKLM\SYSTEM\CurrentControlSet\Control\NetworkProvider\HwOrder\ProviderOrder, чтобы определить, какие провайдеры сетей загружены. Далее он опрашивает их в том порядке, в каком они перечислены в реестре, и делает это до тех пор, пока один из них не распознает сетевой ресурс или пока все они не будут опрошены. Параметр ProviderOrder можно изменить через диалоговое окно Advanced Settings (Дополнительные параметры), показанное на рис. 13-14. (В системе, в которой был сделан этот экранный снимок, установлен только

один провайдер.) В Windows 2000 или при настройке меню Start (Пуск) в традиционном стиле это диалоговое окно вызывается из меню Advanced (Дополнительно) апплета Network Connections (Сетевые подключения), который запускается несколькими способами. Вы можете, например, щелкнуть правой кнопкой мыши значок My Network Places (Мое сетевое окружение) на рабочем столе и выбрать из контекстного меню команду Properties (Свойства), либо открыть меню Start (Пуск), затем подменю Settings (Настройка) и выбрать команду Network Connections (Сетевые подключения).

Функция *WNetAddConnection* может также назначить удаленному ресурсу букву диска или имя устройства. В этом случае она направляет вызов соответствующему компоненту сетевого доступа. Тот в свою очередь создает объект «символьная ссылка» в пространстве имен диспетчера объектов, и этот объект увязывает данную букву диска с редиректором нужной сети (т. е. с удаленным FSD).

На рис. 13-15 показан каталог \?? в системе Windows 2000, в котором вы заметите несколько букв диска, представляющих соединения с удаленными файловыми ресурсами. Как видите, редиректор создает объект «устройство» с именем \Device\LanmanRedirector, а дополнительный текст, который входит в значение символьной ссылки, сообщает редиректору, какому удаленному ресурсу соответствует буква диска. Когда пользователь открывает X:\Book\Chap13.doc, редиректору передается неразобранная часть пути, которая разрешается через символьную ссылку как «;X:0\dual\e\Book\Chap13.doc». Редиректор отмечает, что данный ресурс расположен на общем диске E сервера *dual*.

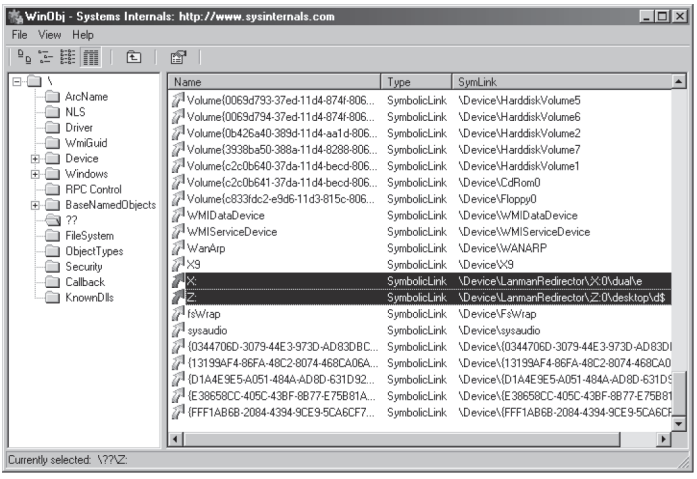


Рис. 13-15. *Разрешение имени сетевого ресурса*

Как и встроенный редиректор, другие редиректоры тоже создают объект «устройство» в пространстве имен диспетчера объектов в процессе своей загрузки и инициализации. После этого, когда WNet или другой API обращается к диспетчеру объектов для открытия ресурса, расположенного в другой

сети, диспетчер использует данный объект «устройство» как точку входа в удаленную файловую систему. Он вызывает метод разбора, принадлежащий диспетчеру ввода-вывода и сопоставленный с объектом, для поиска FSD редиректора, способного обработать данный запрос (о драйверах файловых систем см. главу 12).

Многосетевой UNC-провайдер

Многосетевой UNC-провайдер (Multiple UNC Provider, MUP) — сетевой компонент, сходный с MPR. Он обрабатывает запросы ввода-вывода (адресованные к файлам или устройствам) с UNC-именами (именами, которые начинаются с символов \\, указывающих, что данный ресурс находится в сети). MUP, как и MPR, определяет, какой локальный редиректор распознает удаленный ресурс. Но MUP в отличие от MPR является драйвером устройства (загружаемым при загрузке системы), который выдает запросы на ввод-вывод драйверам более низкого уровня, в данном случае — редиректорам, как показано на рис. 13-16. Mup.sys также содержит клиентскую реализацию Distributed File System (DFS). Клиент DFS включен по умолчанию, и его можно отключить, присвоив DWORD-параметру реестра HKLM\System\CurrentControlSet\Services\Mup\DisableDfs значение 1.

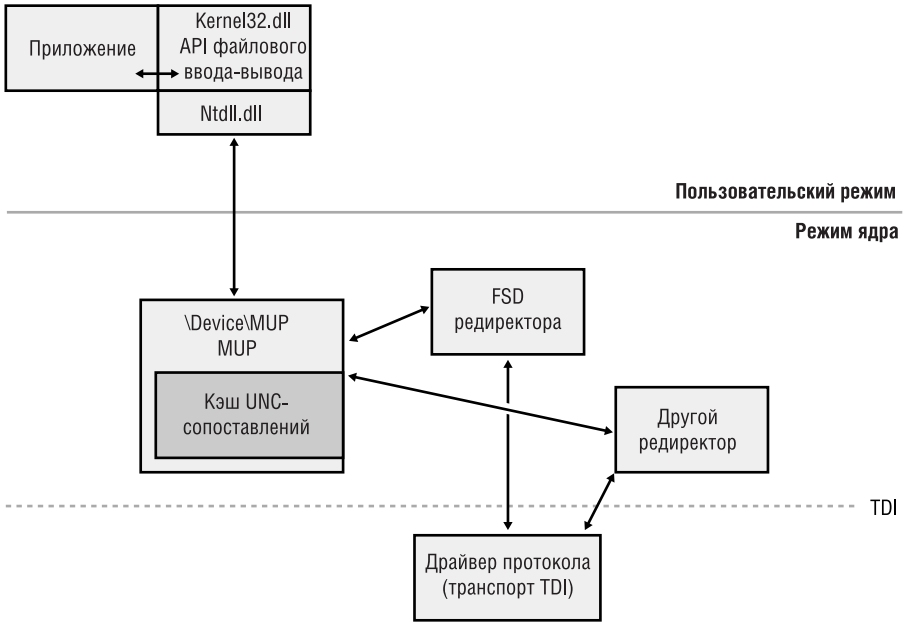


Рис. 13-16. *Многосетевой UNC-провайдер (MUP)*

При загрузке MUP создает объект «устройство» с именем \Device\Mup. Когда сетевой редиректор вроде CIFS загружает редиректор, тот создает именованный объект «устройство» (скажем, \Device\LanmanRedirector) и регистрируется в MUP как UNC-провайдер вызовом функции *FsRtlRegister-*

UncProvider. Если этот редиректор — первый из зарегистрированных и если поддержка DFS-клиента в MUP отключена, то *FsRtlRegisterUncProvider* создает символьную ссылку \??\UNC, которая указывает на объект «устройство» редиректора; в ином случае MUP настраивает символьную ссылку \Global??\UNC (\??\UNC в Windows 2000) так, чтобы она указывала на его объект «устройство», \Device\MUP.

Драйвер MUP активизируется, когда приложение впервые пытается открыть удаленный файл или устройство по UNC-имени (а не по букве сетевого диска). Получив запрос на ввод-вывод с UNC-путем, Kernel32.dll (экспортирующая API-функции файлового ввода-вывода) на клиентской стороне добавляет переданный в запросе UNC-путь к строке \Global??\UNC, после чего вызывает системный сервис *NtCreateFile* для открытия файла.

Если зарегистрирован только один провайдер сети, то \Global??\UNC разрешается в объект «устройство», представляющий драйвер, и запрос обрабатывается этим драйвером. При наличии нескольких зарегистрированных провайдеров \Global??\UNC разрешается в \Device\MUP, и MUP должен определить, какой провайдер будет обрабатывать данный запрос.

Когда драйвер MUP принимает запрос ввода-вывода и клиент DFS включен, MUP сначала определяет, соответствует ли указанный путь DFS-пути (DFS-пути тоже форматируются по стандарту UNC), и, если да, сам обрабатывает запрос. Если клиент DFS отключен или путь не соответствует DFS-пути, MUP считывает параметр реестра HKLM\SYSTEM\CurrentControlSet\Control\NetworkProvider\Order\ProviderOrder, чтобы определить приоритет провайдеров сетей, зарегистрированных через *FsRtlRegisterUncProvider*. Затем MUP поочередно опрашивает провайдеры в том порядке, в каком они перечислены в данном параметре реестра, до тех пор, пока один из них не сообщит, что он распознал данный путь, или пока не будут опрошены все имеющиеся провайдеры. MUP игнорирует те редиректоры, которые указаны в параметре ProviderOrder, но не зарегистрированы. Когда один из редиректоров распознает путь, он сообщает, какая часть пути уникальна именно для него. Например, если путем является строка «\\WIN2K3SERVER\PUBLIC\Windowsinternals\Chap13.doc», редиректор может распознать и объявить своей подстроку «\\WIN2K3SERVER\PUBLIC». Драйвер MUP кэширует эту информацию и впоследствии пересылает запросы, начинающиеся с данной подстроки, непосредственно этому редиректору, пропуская стадию опроса. Кэш драйвера MUP хранит данные в течение определенного периода, поэтому через некоторое время сопоставление подстроки с данным редиректором становится недействительным.

Разрешение имен

Разрешение имен (name resolution) — это процесс, в ходе которого символьное имя вроде Mycomputer или www.microsoft.com транслируется в числовой адрес типа 192.168.1.1, распознаваемый стеком протоколов. В этом разделе описываются два TCP/IP-протокола разрешения имен, предоставляемые Windows, — DNS (Domain Name System) и WINS (Windows Internet Name Service).

DNS

DNS (Domain Name System) — стандарт трансляции имен в Интернете (например, *www.microsoft.com*) в соответствующие IP-адреса. Сетевое приложение, которому требуется разрешить DNS-имя в IP-адрес, использует TCP/IP для передачи серверу запроса на поиск DNS-имени. DNS-серверы реализуют распределенную базу данных сопоставлений имен и IP-адресов, используемых при разрешении. Каждый сервер обслуживает разрешение имен для определенной *зоны*. Подробное описание DNS не входит в задачи этой книги, но DNS представляет собой основной протокол разрешения имен в Windows.

DNS-сервер реализован в виде Windows-сервиса (\Windows\System32\Dns.exe), который входит в состав серверных версий Windows. DNS-сервер в стандартной реализации использует в качестве базы данных текстовый файл, но DNS-сервер в Windows может быть сконфигурирован на хранение зонной информации в Active Directory.

WINS

Сетевая служба WINS (Windows Internet Name Service) хранит и поддерживает сопоставления между NetBIOS-именами и IP-адресами, используемые TCP/IP-приложениями на основе NetBIOS. Если WINS не установлена, NetBIOS разрешает имена, рассылая широковещательные сообщения в локальной подсети. Заметьте, что NetBIOS-имена вторичны по отношению к DNS-именам в случае приложений Windows Sockets: имена компьютеров регистрируются и разрешаются сначала через DNS. Windows возвращается к NetBIOS-именам, только если разрешение имени через DNS заканчивается неудачно.

Драйверы протоколов

Драйверы сетевых API должны принимать запросы, адресованные к API, и транслировать их в низкоуровневые запросы сетевых протоколов для передачи по сети. Драйверы API выполняют реальную трансляцию с помощью драйверов транспортных протоколов в режиме ядра. Отделение API от нижележащих протоколов придает сетевой архитектуре гибкость, позволяющую каждому API использовать множество различных протоколов. В Windows входят следующие драйверы протоколов: TCP/IP, TCP/IP с IPv6, NWLink и AppleTalk. Ниже дается краткое описание каждого из этих протоколов.

- Взрывное развитие Интернета и популярность TCP/IP обусловили статус этих протоколов как основных в Windows. TCP/IP был разработан DARPA (Defense Advanced Research Projects Agency) в 1969 году как фундамент Интернета, поэтому характеристики TCP/IP (поддержка маршрутизации и хорошая производительность в WAN) благоприятствуют его использованию в глобальных сетях. TCP/IP — основной стек протоколов в Windows. Он устанавливается по умолчанию, и его нельзя удалить.

- 4-байтовые сетевые адреса, используемые протоколом IPv4 в стандартном стеке протоколов TCP/IP, ограничивают число общедоступных IP-адресов примерно до 4 миллиардов. И это становится серьезной проблемой, поскольку в Интернете появляется все больше и больше устройств, таких как сотовые телефоны и КПК. По этой причине начинается внедрение протокола IPv6, в котором каждый адрес имеет 16 байтов. В Windows XP (Service Pack 1 и выше) и Windows Server 2003 включен стек TCP/IP, \Windows\System32\Drivers\Tcpip6.sys, реализующий IPv6. Windows-реализация IPv6 совместима с сетями на основе IPv4 за счет туннелирования.
- NWLink состоит из протоколов Novell IPX и SPX. NWLink включен в Windows для взаимодействия с серверами Novell NetWare.
- Протокол AppleTalk используется в сетях Apple Macintosh; его поддержка позволяет Windows взаимодействовать со службами доступа к файлам и принтерам в сетях на основе AppleTalk.

В Windows транспорты TDI в общем случае реализуют все протоколы, сопоставленные с основным стеком протоколов. Например, драйвер TCP/IP IPv4 (\Windows\System32\Drivers\Tcpip.sys) реализует протоколы TCP, UDP, IP, ARP, ICMP и IGMP. Для представления конкретных протоколов транспорт TDI обычно создает объекты «устройство», что позволяет клиентам получать объект «файл», представляющий нужный протокол, и выдавать ему запросы на сетевой ввод-вывод с использованием IRP. Драйвер TCP/IP создает несколько объектов «устройство» для представления различных протоколов, доступных клиентам TDI: \Device\Tcp, \Device\Udp и \Device\Ip, а также (в Windows XP и Windows Server 2003) \Device\Rawip и \Device\Ipmulticast.

ЭКСПЕРИМЕНТ: просмотр объектов «устройство», принадлежащих TCP/IP

С помощью отладчика ядра можно изучить эти объекты в работающей системе. Команда *!drvobj* позволяет узнать адрес каждого объекта «устройство» драйвера, а *!devobj* — просмотреть имя и другие сведения о конкретном объекте.

```
lkd> !drvobj tcpip
Driver object (8a01ada0) is for:
 \Driver\Tcpip
Driver Extension List: (id , addr)

Device Object list:
8a0dbc88  8a0dc958  8a0dcd80  8a0eff18
8a0f32a0

lkd> !devobj 8a0dbc88
Device object (8a0dbc88) is for:
 RawIp \Driver\Tcpip DriverObject 8a01ada0
Current Irp 00000000 RefCount 3 Type 00000012 Flags 00000050
Dacl e100d19c DevExt 00000000 DevObjExt 8a0dbd40
```

```
ExtensionFlags (0000000000)
Device queue is not busy.

lkd> !devobj 8a0dc958
Device object (8a0dc958) is for:
 Udp \Driver\Tcpip DriverObject 8a01ada0
Current Irp 00000000 RefCount 41 Type 00000012 Flags 00000050
Dacl e100d19c DevExt 00000000 DevObjExt 8a0dca10
ExtensionFlags (0000000000)
Device queue is not busy.

lkd> !devobj 8a0dcd80
Device object (8a0dcd80) is for:
 Tcp \Driver\Tcpip DriverObject 8a01ada0
Current Irp 00000000 RefCount 302 Type 00000012 Flags 00000050
Dacl e1c3a1ac DevExt 00000000 DevObjExt 8a0dce38
ExtensionFlags (0000000000)
Device queue is not busy.

lkd> !devobj 8a0eff18
Device object (8a0eff18) is for:
 IPMULTICAST \Driver\Tcpip DriverObject 8a01ada0
Current Irp 00000000 RefCount 0 Type 00000012 Flags 00000040
Dacl e100d19c DevExt 00000000 DevObjExt 8a0effd0
ExtensionFlags (0000000000)
Device queue is not busy.

lkd> !devobj 8a0f32a0
Device object (8a0f32a0) is for:
 Ip \Driver\Tcpip DriverObject 8a01ada0
Current Irp 00000000 RefCount 48 Type 00000012 Flags 00000050
Dacl e1c3a1ac DevExt 00000000 DevObjExt 8a0f3358
ExtensionFlags (0000000000)
Device queue is not busy.
```

Microsoft определила стандарт TDI (Transport Driver Interface), чтобы драйверам сетевых API не приходилось использовать отдельные интерфейсы для каждого необходимого им транспортного протокола. Как уже говорилось, интерфейс TDI фактически представляет собой правила форматирования сетевых запросов в IRP, а также выделения сетевых адресов и коммуникационных соединений. Транспортные протоколы, отвечающие стандарту TDI, экспортируют интерфейс TDI своим клиентам, в число которых входят драйверы сетевых API, например AFD и редиректор. Транспортный протокол, реализованный в виде драйвера устройства Windows, называется транспортом TDI. Поскольку транспорты TDI являются драйверами устройств, они преобразуют получаемые от клиентов запросы в формат IRP.

Интерфейс TDI образуют функции поддержки из библиотеки \Windows\System32\Drivers\Tdi.sys вместе с определениями, включаемыми разработчи-

ками в свои драйверы. Модель программирования TDI очень напоминает таковую в Winsock. Устанавливая соединение с удаленным сервером, клиент TDI выполняет следующие действия.

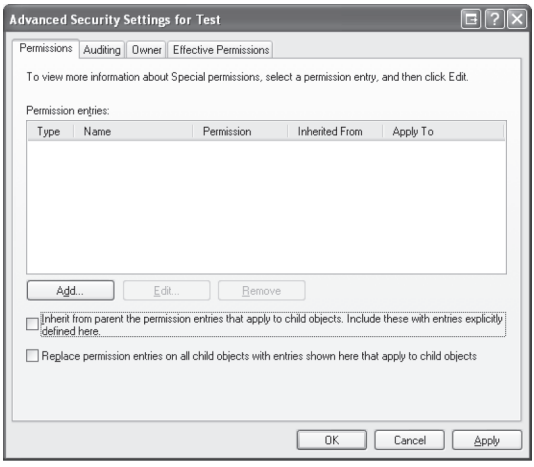
1. Чтобы выделить адрес, клиент создает и форматирует TDI IRP-пакет *address open*. Транспорт TDI возвращает объект «файл», который представляет адрес и называется объектом адреса (address object). Эта операция эквивалентна вызову Winsock-функции *bind*.
2. Далее клиент создает и форматирует TDI IRP-пакет *connection open*, а транспорт TDI возвращает объект «файл», который представляет соединение и называется объектом соединения (connection object). Эта операция эквивалентна вызову Winsock-функции *socket*.
3. Клиент сопоставляет объект соединения с объектом адреса с помощью TDI IRP-пакета *associate address* (для этой операции эквивалентных функций Winsock нет).
4. Клиент TDI, соглашающийся установить удаленное соединение, выдает TDI IRP-пакет *listen*, указывая для объекта соединения максимальное число подключений. После этого он выдает TDI IRP-пакет *accept*, обработка которого заканчивается либо установлением соединения с удаленной системой, либо ошибкой. Эти операции эквивалентны вызову Winsock-функций *listen* и *accept*.
5. Клиент TDI, которому нужно установить соединение с удаленным сервером, выдает TDI IRP-пакет *connect*, указывая объект соединения, выполняемый транспортом TDI после установления соединения или появления ошибки. Выдача TDI IRP-пакета *connect* эквивалентна вызову Winsock-функции *connect*.

TDI также поддерживает коммуникационную связь, не требующую логических соединений, для протоколов соответствующего типа, например для UDP. Кроме того, TDI предоставляет клиенту TDI средства для регистрации в транспортах TDI своих функций обратного вызова по событиям (event callbacks) (т. е. функций, вызываемых напрямую). Например, при получении данных через сеть транспорт TDI может вызвать зарегистрированную клиентом функцию обратного вызова для приема данных. Поддержка функций обратного вызова на основе событий позволяет транспорту TDI уведомлять своих клиентов о сетевых событиях, а клиенты, использующие такие функции, могут не выделять ресурсы для приема данных из сети, поскольку им доступно содержимое буферов, предоставляемых драйвером протокола TDI.

ЭКСПЕРИМЕНТ: наблюдаем активность, связанную с TDI

Утилита TDImon (*wwwsysinternals.com*) является разновидностью драйвера фильтра, который подключается к объектам «устройство» \Device\Tcp и \Device\Udp, создаваемым драйвером TCP/IP. После подключения TDImon может наблюдать за каждым IRP, выдаваемым клиентами TDI своим протоколам. TDImon также может отслеживать функ-

ции обратного вызова по событиям, перехватывая запросы на их регистрацию от клиентов TDI. Драйвер TDImon посылает информацию об активности TDI своему графическому пользовательскому интерфейсу, который и отображает эти сведения (время операции, тип активности TDI, локальный и удаленный адреса TCP-соединения или локальный адрес конечной точки UDP, код статуса IRP и др.). Ниже приведен экранный снимок окна TDImon, в котором ведется мониторинг активности TDI при просмотре Web-страницы в Internet Explorer.



Как доказательство «врожденной» асинхронности операций TDI, в колонке Result выводятся сообщения «PENDING». Это говорит о том, что операция инициирована, но обработка IRP, вызвавшего ее выполнение, еще не завершена. Чтобы было видно, в каком порядке одни операции завершаются относительно начала других, факт выдачи каждого IRP или обращения к функции обратного вызова отмечается своим порядковым номером. Если до завершения обработки данного IRP генерируются или завершаются другие IRP, эти факты также отмечаются соответствующими порядковыми номерами, которые показываются в колонке Result. Например, на нашей иллюстрации IRP 1278 завершился после генерации IRP 1279, поэтому в колонке Result для IRP 1278 выводится число 1280.

Расширения TCP/IP

Ряд сетевых сервисов Windows расширяет базовые сетевые возможности драйвера TCP/IP за счет применения драйверов-надстроек, интегрируемых с драйвером TCP/IP через закрытые интерфейсы. К числу таких сервисов относятся трансляция сетевых адресов (NAT), IP-фильтрация, подключение IP-ловушек (IP-hooking) и IP-безопасность (IPSec). На рис. 13-17 показано, как эти расширения связаны с драйвером TCP/IP.

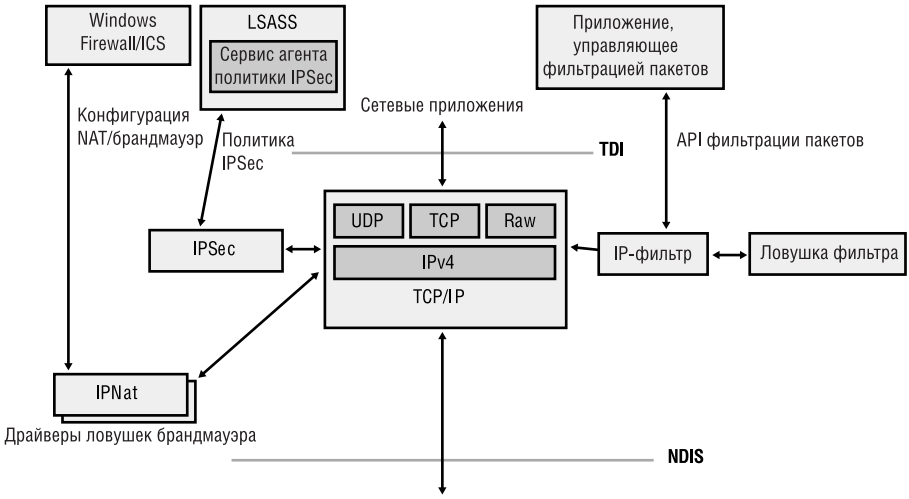


Рис. 13-17. *Архитектура расширений TCP/IP*

Трансляция сетевых адресов

Трансляция сетевых адресов (network address translation, NAT) представляет собой сервис маршрутизации, позволяющий отображать несколько закрытых IP-адресов на один общий IP-адрес, видимый в Интернете. Без NAT для коммуникационной связи с Интернетом каждому компьютеру в локальной сети (LAN) пришлось бы назначать свой IP-адрес, видимый в Интернете. NAT дает возможность назначить такой IP-адрес только одному из компьютеров в локальной сети и подключать остальные компьютеры к Интернету через него. NAT по мере необходимости транслирует LAN-адреса в общий IP-адрес, перенаправляя пакеты из Интернета на соответствующий компьютер в локальной сети.

Компоненты NAT в Windows — драйвер устройства NAT (\Windows\System32\Drivers\Ipnat.sys), взаимодействующий со стеком TCP/IP, а также редакторы, с помощью которых возможна дополнительная обработка пакетов (помимо трансляции адресов и портов). NAT может быть установлен как маршрутизирующий протокол через оснастку Routing And Remote Access (Маршрутизация и удаленный доступ) консоли MMC или настройкой Интернет-соединения на общее использование через апплет Network Connections (Сетевые подключения). (Более широкие возможности в настройке NAT предоставляет оснастка Routing And Remote Access.)

IP-фильтрация

В Windows 2000, Windows XP и Windows Server 2003 есть минимальные базовые средства IP-фильтрации, позволяющие пропускать пакеты только по определенным портам или IP-протоколам. Хотя эти средства в какой-то мере защищают компьютер от несанкционированного доступа из сети, их недо-

статок в том, что они статичны и не предусматривают возможность автоматического создания новых фильтров для трафика, инициируемого работающими на компьютере приложениями.

В Windows XP введен персональный брандмауэр — Windows Firewall, возможности которого шире, чем у базовых средств фильтрации. Windows Firewall реализует брандмауэр с поддержкой состояний (stateful firewall), который отслеживает и различает трафик, генерируемый TCP/IP, и трафик, поступающий из LAN и Интернета. Когда вы включаете Windows Firewall для какого-либо сетевого интерфейса, весь незатребованный входящий трафик по умолчанию отбрасывается. Приложение или пользователь может определить исключения, чтобы сервисы, работающие на данном компьютере (вроде службы доступа к общим файлам и принтерам), были доступны с других компьютеров.

Сервис Windows Firewall/ICS (Internet Connection Sharing), выполняемый в процессе Svchost, передает правила исключения, определенные через пользовательский интерфейс Windows Firewall, драйверу IPNat. В режиме ядра Windows Firewall реализован в том же драйвере (\Windows\System32\Drivers\Ipnat.Sys), который реализует трансляцию сетевых адресов (NAT). Драйвер NAT регистрируется в драйвере TCP/IP как драйвер *ловушки брандмауэра* (firewall hook). Драйвер TCP/IP выполняет функции обратного вызова каждой зарегистрированной ловушки брандмауэра в ходе обработки входящих и исходящих IP-пакетов. Функция обратного вызова может выступать в роли NAT, модифицируя адреса источника и получателя в пакете, или в роли брандмауэра, возвращая код состояния, указывающий TCP/IP отбросить пакет.

Хотя Ipnat реализует Windows Firewall с применением интерфейса ловушек брандмауэра TCP/IP, Microsoft рекомендует сторонним разработчикам реализовать поддержку фильтрации пакетов в виде промежуточного драйвера NDIS (о нем мы еще расскажем в этой главе).

IP-фильтр и ловушка фильтра

В Windows XP и Windows Server 2003 включен API фильтрации пакетов пользовательского режима, а также драйвер фильтра IP, \Windows\System32\Drivers\Ipfltrdrv.sys, которые позволяют приложениям управлять входящими и исходящими пакетами. Кроме того, драйвер фильтра IP дает возможность максимум одному драйверу регистрироваться в качестве драйвера *ловушки фильтра* (filter hook). TCP/IP — по аналогии с тем, как он взаимодействует с драйверами ловушек брандмауэра, — выполняет функцию, которую указывает драйвер фильтра IP, а это позволяет IP-фильтру отбрасывать или модифицировать пакеты. В свою очередь IP-фильтр обращается к функции обратного вызова, заданной драйвером ловушки фильтра, и тем самым передает изменения или запрос на отклонение пакета драйверу TCP/IP.

Функциональность ловушки фильтра, предоставляемая системой, дает возможность сторонним разработчикам добавлять новые средства трансляции, брандмауэра, протоколирования и т. д.

IP-безопасность

IP-безопасность (Internet Protocol Security, IPSec) интегрирована со стеком TCP/IP и защищает одноадресные (unicast) IP-данные от перехвата и несанкционированной модификации, подмены IP-адресов и атак через посредника (man-in-the-middle attacks). IPSec обеспечивает глубоко эшелонированную оборону от сетевых атак с недоверяемых компьютеров, от атак, которые могут привести к отказу в обслуживании, от повреждения данных, кражи информации и учетных данных пользователей, а также от попыток захватить административный контроль над серверами, другими компьютерами и сетью. Эти цели реализуются за счет сервисов защиты на основе шифрования, протоколов безопасности и динамического управления ключами. При обмене одноадресными IP-пакетами между доверяемыми хостами IPSec поддерживает следующую функциональность:

- аутентификацию источника данных — проверку источника IP-пакета и запрет несанкционированного доступа к данным;
- целостность данных — защиту IP-пакета от модификации в процессе доставки и распознавание любых изменений;
- конфиденциальность — содержимое IP-пакетов зашифровывается перед отправкой, благодаря чему их содержимое может быть расшифровано только указанным адресатом;
- защиту от повторений пакетов (anti-replay, или replay protection) — гарантирует уникальность каждого IP-пакета и невозможность его повторного использования. Злоумышленник, даже если он сумеет перехватить IP-пакеты, не сможет повторно использовать их для установления сеанса связи или неавторизованного доступа к информации.

Для защиты от сетевых атак вы можете настроить IPSec на фильтрацию пакетов хостом (host-based packet filtering) и разрешать соединения только с доверяемыми компьютерами. После настройки на фильтрацию пакетов хостом IPSec может разрешать или блокировать определенные виды одноадресного IP-трафика, исходя из адресов источника и получателя, заданных протоколов и портов. Поскольку IPSec интегрирован с IP-уровнем (уровнем 3) стека TCP/IP и действует на все приложения, вам не понадобится настраивать параметры безопасности индивидуально для каждого приложения, работающего с TCP/IP.

В среде Active Directory групповая политика позволяет настраивать домены, сайты и организационные единицы (organizational units, OU), а политики IPSec можно закреплять за нужными объектами групповой политики (Group Policy Objects, GPO). В качестве альтернативы можно конфигурировать и применять локальные политики IPSec. Политики IPSec хранятся в Active Directory, а копия параметров текущей политики поддерживается в кэше в локальном реестре. Локальные политики IPSec хранятся в реестре локальной системы.

Для установления доверяемого соединения IPSec использует взаимную аутентификацию (mutual authentication), при этом поддерживаются следу-

ющие методы аутентификации: Kerberos версии 5, сертификат открытого ключа X.509 версии 3 или на основе общего ключа (preshared key).

Windows-реализация IPSec основана на RFC, относящихся к IPSec. Архитектура Windows IPSec включает IPSec Policy Agent (Агент политики IP-безопасности), протокол Internet Key Exchange (IKE) и драйвер IPSec.

- **Агент политики IP-безопасности** Выполняется как сервис в процессе LSASS (о LSASS см. главу 8). В MMC-оснастке Services (Службы) в списке служб он отображается как IPSEC Services (Службы IPSEC). Агент политики IP-безопасности получает политику IPSec из домена Active Directory или из локального реестра и передает фильтры IP-адресов драйверу IPSec, а параметры аутентификации и безопасности — IKE.
- **IKE** Ожидает от драйвера IPSec запросы на согласование *сопоставлений безопасности* (security associations, SA), согласовывает SA, а потом возвращает параметры SA драйверу IPSec. SA — это набор взаимно согласованных параметров политики IPSec и ключей, определяющий службы и механизмы защиты, которые будут использоваться при защищенной коммуникационной связи между двумя равноправными хостами с IPSec. Каждое SA является односторонним, или симплексным, соединением, которое защищает передаваемый по нему трафик. IKE согласует SA основного и быстрого режимов, когда от драйвера IPSec поступает соответствующий запрос. SA основного режима IKE (или ISAKMP) защищает процесс согласования, выполняемый IKE, а SA быстрого режима (или IPSec) — трафик приложений.
- **Драйвер IPSec** Это драйвер устройства (\Windows\System32\Drivers\Ipsec.sys), который привязывается к драйверу TCP/IP и который обрабатывает пакеты, передаваемые через драйвер TCP/IP. Драйвер IPSec отслеживает и защищает исходящий одноадресный IP-трафик, а также отслеживает, расшифровывает и проверяет входящие одноадресные IP-пакеты. Этот драйвер принимает фильтры от агента политики IP-безопасности, а затем пропускает, блокирует или защищает пакеты в соответствии с критериями фильтров. Для защиты трафика драйвер IPSec использует параметры активного SA либо запрашивает создание новых SA.

MMC-оснастка IP Security Policy Management (Управление политикой безопасности IP) позволяет создавать политику IPSec и управлять ею. С помощью этой оснастки можно создавать, изменять и сохранять локальные политики IPSec или политики IPSec на основе Active Directory, а также модифицировать политику IPSec на удаленных компьютерах. В Windows XP и Windows Server 2003, после того как защищенное IPSec-соединение установлено, вы можете отслеживать информацию IPSec для локального и удаленных компьютеров через MMC-оснастку IP Security Monitor (Монитор IP-безопасности).

Драйверы NDIS

Когда драйверу протокола требуется получить или отправить сообщение в формате своего протокола, он должен сделать это с помощью сетевого адап-

тера. Поскольку ожидать от драйверов протоколов понимания нюансов работы каждого сетевого адаптера нереально (на рынке предлагается несколько тысяч моделей сетевых адаптеров с закрытой спецификацией), производители сетевых адаптеров предоставляют драйверы устройств, которые принимают сетевые сообщения и передают их через свои устройства. В 1989 году компании Microsoft и 3Com совместно разработали спецификацию Network Driver Interface Specification (NDIS), которая определяет аппаратно-независимое взаимодействие драйверов протоколов с драйверами сетевых адаптеров. Драйверы сетевых адаптеров, соответствующие NDIS, называются драйверами NDIS или минипорт-драйверами NDIS. С Windows 2000 поставляется NDIS версии 5, а с Windows XP и Windows Server 2003 — версии 5.1.

Библиотека NDIS (\Windows\System32\Drivers\Ndis.sys) реализует пограничный уровень между транспортами TDI (в типичном случае) и драйверами NDIS. Как и Tdi.sys, библиотека NDIS является вспомогательной и используется клиентами драйверов NDIS для форматирования команд, посылаемых этим драйверам. Драйверы NDIS взаимодействуют с библиотекой, чтобы получать запросы и отвечать на них. Взаимосвязи между компонентами, имеющими отношение к NDIS, показаны на рис. 13-18.

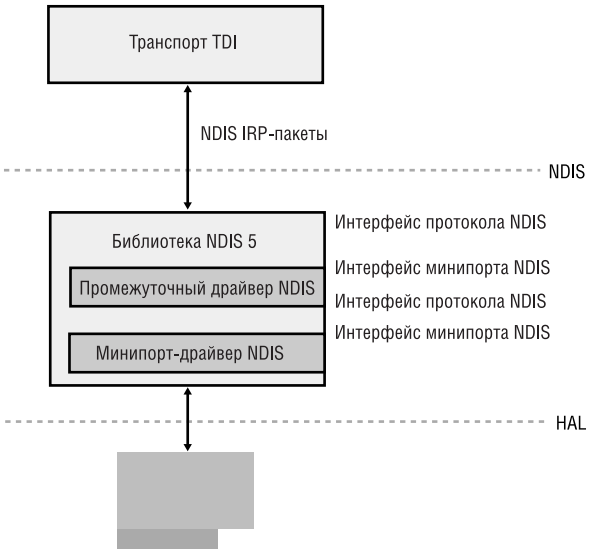


Рис. 13-18. *Компоненты NDIS*

Одна из целей Microsoft при разработке сетевой архитектуры состояла в том, чтобы производителям сетевых адаптеров было легче разрабатывать драйверы NDIS и переносить их код между потребительскими версиями Windows и Windows 2000. Таким образом, библиотека NDIS предоставляет драйверам не просто вспомогательные пограничные процедуры NDIS, а целую среду выполнения драйверов NDIS. Последние не являются истинными драйверами Windows, поскольку не могут функционировать без инкапсули-

рующей их библиотеки NDIS. Этот инкапсулирующий уровень является настолько плотной оболочкой драйверов NDIS, что они не принимают и не обрабатывают IRP. Вместо этого драйверы протоколов TDI вызывают функцию *NdisAllocatePacket* в библиотеке NDIS и передают пакеты минипорту NDIS, вызывая соответствующую NDIS-функцию, например *NdisSend*. По умолчанию драйверам NDIS также не приходится заботиться о реентерабельности, когда библиотека NDIS вызывает драйвер с новым запросом до того, как он успел обработать предыдущий запрос. Освобождение от поддержки реентерабельности кода означает, что создатели драйверов NDIS могут не думать о сложных проблемах синхронизации, которые еще больше усложняются в многопроцессорных системах.

ПРИМЕЧАНИЕ Библиотека NDIS использует для представления запросов ввода-вывода NDIS-пакеты, а не IRP. Транспорты TDI создают NDIS-пакет вызовом *NdisAllocatePacket*, после чего пакет передается минипорту NDIS вызовом одной из функций библиотеки NDIS (например, *NdisSend*).

Хотя сериализация обращений к драйверам NDIS, осуществляемая библиотекой NDIS, упрощает разработку, она может помешать масштабированию многопроцессорных систем. Некоторые операции стандартных драйверов NDIS 4 (версия библиотеки NDIS 4 из Windows NT 4) плохо масштабируются в многопроцессорных системах. В NDIS 5 разработчики получили возможность отказаться от такой сериализации. Драйвер NDIS 5 может сообщить библиотеке NDIS, что сериализация ему не нужна, и тогда библиотека NDIS переправляет драйверу запросы по мере получения соответствующих IRP. В этом случае ответственность за управление параллельными запросами ложится на драйвер NDIS, но отказ от сериализации окупается повышением производительности в многопроцессорных системах.

NDIS 5 также обеспечивает следующие преимущества.

- Драйверы NDIS могут сообщать, активна ли несущая сетевая среда, что позволяет Windows выводить на панель задач значок, показывающий, подключен ли компьютер к сети. Эта функция также позволяет протоколам и другим приложениям быть в курсе этого состояния и соответствующим образом реагировать. Например, транспорт TCP/IP будет использовать эту информацию, чтобы определять, когда нужно заново оценивать информацию об адресах, получаемую им от DHCP.
- Аппаратное ускорение TCP/IP-операций (TCP/IP task offload) позволяет минипорту пользоваться аппаратными функциями сетевого адаптера для выполнения таких операций, как расчет контрольных сумм пакетов и все вычисления, связанные с IP-безопасностью (IPSec). Аппаратное ускорение этих операций средствами сетевого адаптера повышает производительность системы, освобождая центральный процессор от выполнения этих задач.
- Функция Wake-On-LAN дает возможность сетевому адаптеру с соответствующей поддержкой выводить систему Windows из состояния с низким

энергопотреблением при каких-либо событиях в сети. Сигнал пробуждения может быть инициирован сетевым адаптером при одном из следующих событий: подключении к несущей среде (например, подключении сетевого кабеля к адаптеру) и приеме специфичных для протокола последовательностей байтов (в случае адаптеров Ethernet — при получении волшебного пакета, т. е. сетевого пакета с 16 копиями Ethernet-адреса адаптера подряд).

- NDIS, ориентированная на логические соединения, позволяет драйверам NDIS управлять несущей средой, требующей логических соединений, например устройствами ATM (Asynchronous Transfer Mode).

Интерфейсы, предоставляемые библиотекой NDIS драйверам NDIS для взаимодействия с сетевыми адаптерами, доступны через функции, вызовы которых транслируются непосредственно в вызовы соответствующих HAL-функций.

ЭКСПЕРИМЕНТ: перечисление загруженных минипортов NDIS

Библиотека расширения отладчика ядра Ndiskd поддерживает команды *!miniports* и *!miniport*, которые позволяют с помощью отладчика ядра перечислять загруженные минипорт-драйверы (минипорты) и получать детальную информацию о каком-либо минипорте по заданному адресу блока минипорта (структуры данных, применяемой Windows для отслеживания минипортов). Ниже приведен пример использования команд *!miniports* и *!miniport* для вывода списка всех минипорт-драйверов, а также для исследования специфики минипорта, отвечающего за взаимодействие системы с PCI-адаптером Ethernet (заметьте, что WAN-минипорты работают с соединениями удаленного доступа).

```
kd> .load ndiskd
Loaded ndiskd extension DLL

kd> !miniports
Driver verifier level: 0
Failed allocations: 0
Miniport Driver Block: 817aa610
  Miniport: 817b1130 RAS Async Adapter
Miniport Driver Block: 81a1ef30
  Miniport: 81a1ea70 Direct Parallel
Miniport Driver Block: 81a21cd0
  Miniport: 81a217f0 WAN Miniport (PPTP)
 Miniport Driver Block: 81a23290
  Miniport: 81a22130 WAN Miniport (L2TP)
Miniport Driver Block: 81a275f0
  Miniport: 81a25130 Intel 8255x-based PCI Ethernet Adapter (10/100)

kd> !miniport 81a25130
```

```
    Flags          : 20413208
                     BUS_MASTER, INDICATES_PACKETS, IGNORE_REQUEST_QUEUE
                     IGNORE_TOKEN_RING_ERRORS, NDIS_5_0,
                     RESOURCES_AVAILABLE, DESERIALIZED, MEDIA_CONNECTED,
                     NOT_SUPPORTS_MEDIA_SENSE,
    PnPFlags       : 00010021
                     PM_SUPPORTED, DEVICE_POWER_ENABLED, RECEIVED_START
    CheckforHang interval: 2 seconds
        CurrentTick    : 0001
        IntervalTicks  : 0001
    InternalResetCount : 0000
    MiniportResetCount : 0000

    References: 3
    UserModeOpenReferences: 0

    PnPDeviceState : PNP_DEVICE_STARTED
    CurrentDevicePowerState : PowerDeviceD0
    Bus PM capabilities
DeviceD1:                1
DeviceD2:                1
WakeFromD0:              0
WakeFromD1:              1
WakeFromD2:              0
WakeFromD3:              0

SystemState              DeviceState
PowerSystemUnspecified   PowerDeviceUnspecified
S0                       D0
S1                       D1
S2                       PowerDeviceUnspecified
S3                       PowerDeviceUnspecified
S4                       D3
S5                       D3
SystemWake: S1
DeviceWake: D1

WakeupMethodes Enabled 6:
WAKE_UP_PATTERN_MATCH  WAKE_UP_LINK_CHANGE
WakeUpCapabilities of the miniport
MinMagicPacketWakeUp: 4
MinPatternWakeUp: 4
MinLinkChangeWakeUp: 4

    Current PnP and PM Settings:          : 00000030
                     DISABLE_WAKE_UP, DISABLE_WAKE_ON_RECONNECT,
```

см. след. стр.

```
Allocated Resources:
    Memory: f4100000, Length: 1000
    IO Port: 1440, Length: 40
    Memory: f4000000, Length: 100000
    Interrupt Level: 9, Vector: 9
Translated Allocated Resources:
    Memory: f4100000, Length: 1000
    IO Port: 1440, Length: 40
    Memory: f4000000, Length: 100000
    Interrupt Level: 12, Vector: 39
MediaType      : 802.3
DeviceObject   : 81a25030, PhysDO : 81a93cd0  Next DO: 81a63030
MapRegisters   : 819fc000
FirstPendingPkt: 0
SingleWorkItems:
  [0]: 81a254e8 [1]: 81a254f4 [2]: 81a25500 [3]: 81a2550c
  [4]: 81a25518 [5]: 81a25524
DriverVerifyFlags       : 00000000
Miniport Open Block Queue:
  8164b888: Protocol 816524a8 = NBF, ProtocolContext 81649030
  8191f628: Protocol 81928d88 = TCPIP, ProtocolContext 8191f728
 Miniport Interrupt 81a00970
```

Поле Flags исследуемого минипорта показывает, что он поддерживает несериализованные операции (DESERIALIZED), что несущая среда в данный момент активна (MEDIA_CONNECTED) и что он является минипортом NDIS 5 (NDIS_5_0). Кроме того, выводится информация, отражающая сопоставления состояний электропитания устройства и системы, а также список ресурсов шины, назначенных адаптеру диспетчером Plug and Play. (Подробнее о соответствии состояний электропитания устройств и системы см. раздел «Диспетчер электропитания» главы 9.)

Разновидности минипорт-драйверов NDIS

Модель NDIS также поддерживает гибридные NDIS-драйверы транспорта TDI, называемые *промежуточными драйверами NDIS* (NDIS intermediate drivers). Они размещаются между транспортами TDI и драйверами NDIS. Драйверу NDIS промежуточный драйвер кажется транспортом TDI, а транспорту TDI — драйвером NDIS. Промежуточные драйверы NDIS видят весь сетевой трафик в системе, поскольку они расположены между драйверами протоколов и сетевыми драйверами. Программное обеспечение, предоставляющее сетевым адаптерам поддержку отказоустойчивости и балансировки нагрузки, например Microsoft Network Load Balancing Provider, основано на использовании промежуточных драйверов NDIS.

NDIS, ориентированная на логические соединения

Поддержка сетевого оборудования, ориентированного на логические соединения (например, ATM) в Windows является встроенной, и соответствующие стандарты учтены в сетевой архитектуре Windows. Драйверы NDIS, ориентированные на логические соединения, используют многие API, применяемые и стандартными драйверами NDIS, но посылают пакеты через установленные сетевые соединения, а не просто помещают их в сетевую среду.

Кроме поддержки минипорт-драйверов для сетевых сред, ориентированных на логические соединения, в NDIS 5 включены определения для драйверов, поддерживающих такие минипорт-драйверы.

- Диспетчеры вызовов (call managers) являются драйверами NDIS, которые предоставляют сервисы настройки и завершения вызовов для клиентов, ориентированных на логические соединения (см. ниже). Диспетчер вызовов использует ориентированный на логические соединения минипорт, чтобы обмениваться сигнальными сообщениями с другими сетевыми компонентами (аппаратными или программными), например с коммутаторами или другими диспетчерами вызовов. Диспетчер вызовов поддерживает один или несколько сигнальных протоколов вроде ATM User-Network Interface (UNI) 3.1.
- Интегрированный Miniport Call Manager (MCM) представляет собой минипорт-драйвер, ориентированный на логические соединения, который также предоставляет клиентам, требующим логических соединений, сервисы диспетчера вызовов. В сущности, MCM — это минипорт-драйвер NDIS со встроенным диспетчером вызовов.
- Ориентированный на логические соединения клиент использует сервисы настройки и завершения вызовов, предоставляемые диспетчером вызовов или MCM, а также передает и принимает обращения к сервисам минипорт-драйвера NDIS, ориентированного на логические соединения. Такой клиент может предоставлять собственные сервисы протокола более высоким уровням сетевого стека или реализовать уровень эмуляции для взаимодействия с унаследованными протоколами, не требующими логических соединений, и соответствующей несущей средой. Пример уровня эмуляции, реализуемой ориентированным на логические соединения клиентом, — LAN Emulation (LANE), которая скрывает от вышележащих протоколов особенности ориентированной на логические соединения ATM и эмулирует для них несущую среду, не требующую соединений (например, Ethernet).

  Взаимосвязи между этими компонентами показаны на рис. 13-19.

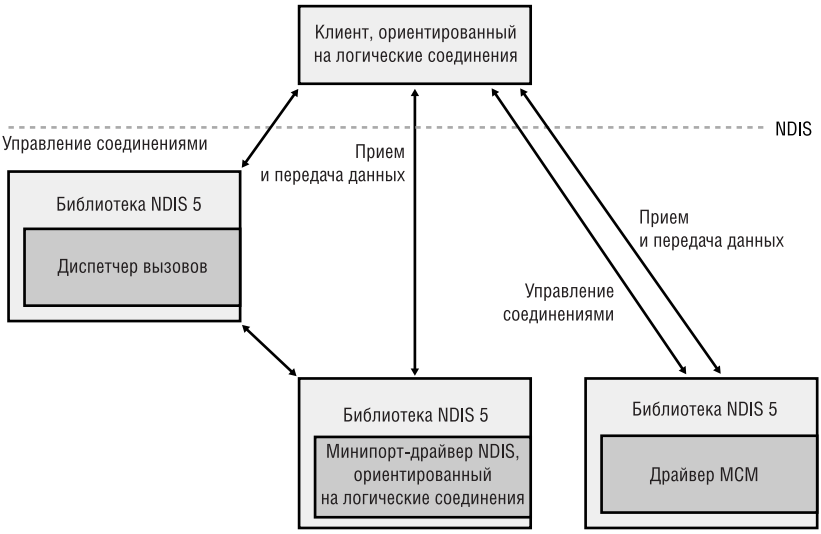
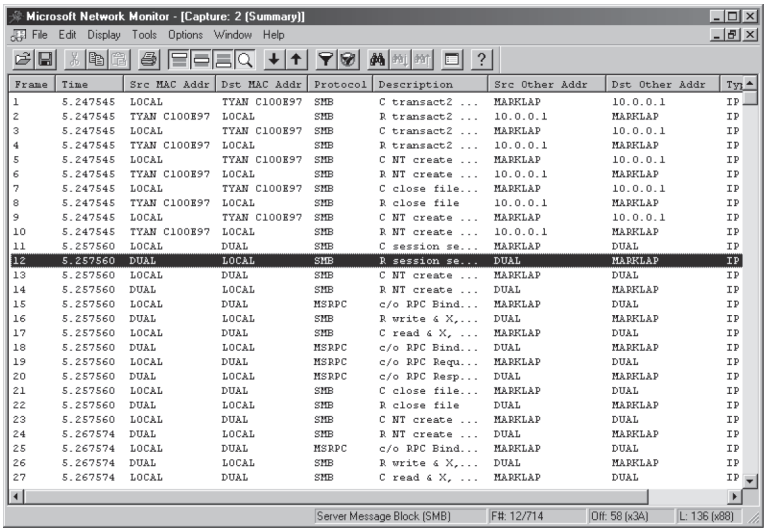


Рис. 13-19. *Драйверы NDIS, ориентированные на логические соединения*

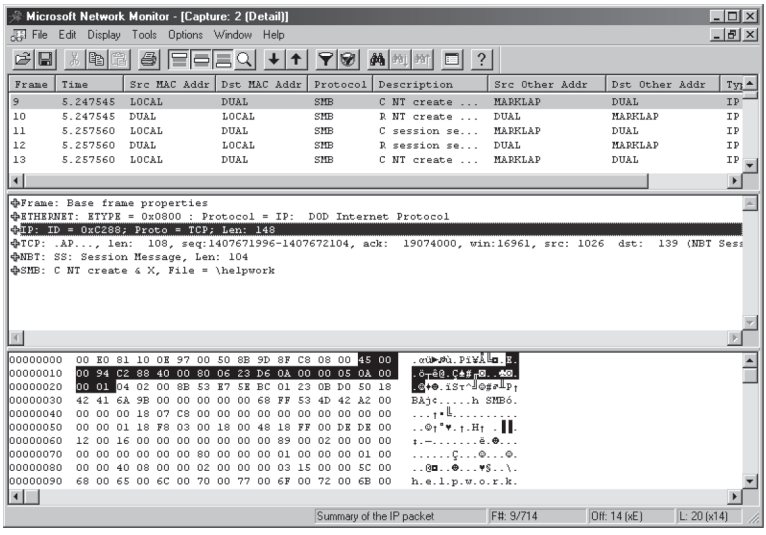
ЭКСПЕРИМЕНТ: захват сетевых пакетов с помощью сетевого монитора

Windows Server поставляется с программой Network Monitor (Сетевой монитор), которая позволяет перехватывать пакеты, проходящие через один или несколько минипорт-драйверов NDIS, за счет установки промежуточного драйвера NDIS. Для использования Network Monitor нужно сначала установить Network Monitor Tools (Средства сетевого монитора). Для этого откройте апплет Add/Remove Programs (Установка и удаление программ) в Control Panel (Панель управления) и выберите Add/Remove Windows Components (Добавление и удаление компонентов Windows). Укажите строку Management And Monitoring Tools (Средства управления и наблюдения), щелкните кнопку Details (Состав), установите флажок в строке Network Monitor Tools (Средства сетевого монитора) и щелкните кнопку OK. После установки Network Monitor (Сетевой монитор) можно запустить, выбрав одноименную команду из меню Administrative Tools (Администрирование).

Network Monitor может спросить, за каким сетевым соединением вы хотите наблюдать. Выбрав нужное соединение, можно начать мониторинг, щелкнув на панели инструментов кнопку Start Capture (Начать запись данных). Выполните несколько операций, генерирующих сетевую активность в отслеживаемом соединении. Увидев, что Network Monitor захватил пакеты, остановите мониторинг, щелкнув кнопку Stop And View Capture (Закончить запись и отобразить данные). В результате Network Monitor покажет захваченные данные.



На этой иллюстрации показаны пакеты SMB (CIFS), захваченные Network Monitor при обращении системы к удаленным файлам. Если вы дважды щелкнете какую-нибудь строку, Network Monitor переключится в режим отображения пакетов, в котором показывается содержимое различных заголовков в пакетах.



Network Monitor поддерживает и другие возможности, например захват триггеров и фильтров, что делает его мощным диагностическим инструментом, помогающим выявлять и устранять неполадки в сети.

Remote NDIS

До разработки Remote NDIS производитель, например, сетевого USB-устройства должен был предоставлять драйвер, который создавал интерфейс с NDIS (в качестве минипорт-драйвера) и с WDM-драйвером шины USB (рис. 13-20). Если производитель оборудования поддерживал другие шины, скажем, IEEE 1394, он должен был реализовать драйверы, создающие интерфейсы с каждым типом шины.

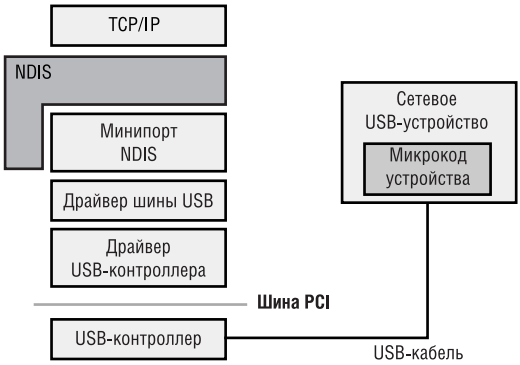


Рис. 13-20. *Минипорт-драйвер NDIS для сетевого USB-устройства*

Remote NDIS — спецификация для сетевых устройств на PnP-шинах ввода-вывода, допускающих динамическое подключение устройств, например USB, IEEE 1394 и Infiniband. Эта спецификация вообще избавляет производителя оборудования от необходимости писать минипорт-драйвер NDIS, так как в ней определены сообщения, независимые от конкретной шины, и механизм, с помощью которого сообщения передаются по различным шинам. В Remote NDIS включены сообщения для инициализации и сброса состояния устройства, передачи и приема пакетов, установки и опроса параметров устройства, а также для уведомления о состоянии канала передачи данных.

Архитектура Remote NDIS (рис. 13-21) построена на минипорт-драйвере NDIS от Microsoft, \Windows\System32\Drivers\Rndismp.sys, который транслирует NDIS-команды и передает их драйверу транспорта для шины, к которой подключено устройство. Эта архитектура позволяет использовать один минипорт-драйвер NDIS для всех драйверов Remote NDIS и один драйвер транспорта для каждой поддерживаемой шины.

В настоящее время Remote NDIS для USB-устройств поддерживается в Windows XP и Windows Server 2003; кроме того, соответствующие компоненты можно скачать с сайта Microsoft и установить в Windows 2000. Хотя Remote NDIS для устройств IEEE 1394 полностью определен, он пока не поддерживается в Windows.

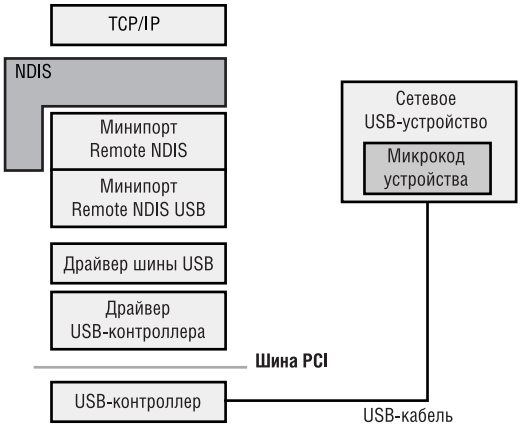


Рис. 13-21. *Архитектура Remote NDIS для сетевых USB-устройств*

QoS

Без специальных мер IP-трафик в сети доставляется по принципу «первым пришел — первым обслужен». Приложения не могут контролировать приоритет своих сообщений, и их данные передаются неравномерно: иногда они получают широкую полосу пропускания и малые задержки, а в остальное время — узкую полосу пропускания и длительные задержки. Хотя такой уровень обслуживания в большинстве ситуаций вполне приемлем, все большее число сетевых приложений требует гарантированных уровней обслуживания, или гарантий *качества обслуживания* (Quality of Service, QoS). Примерами приложений, требующих хорошей сетевой производительности, могут служить видеоконференции, потоковая передача мультимедийной информации и программное обеспечение для планирования ресурсов предприятия (enterprise resource planning, ERP). QoS позволяет приложениям указывать минимальную ширину полосы пропускания и максимальные задержки, которые могут быть удовлетворены только в том случае, если все сетевое программно-аппаратное обеспечение на пути между отправителем и получателем поддерживает стандарты QoS, например IEEE 802.1p — промышленный стандарт, который определяет формат пакетов QoS и реакцию на их получение устройств второго сетевого уровня (коммутаторов и сетевых адаптеров).

Поддержка QoS в Windows основана на наборе Winsock-функций, определенных Microsoft и позволяющих приложениям запрашивать QoS для трафика через свои сокеты Winsock, а также на API управления трафиком (TC API), который позволяет административным приложениям более точно контролировать трафик через сети.

Центральное место в реализации QoS в Windows занимает протокол RSVP (Resource Reservation Setup Protocol), представляющий собой Windows-сервис (\Windows\System32\Rsvp.exe), как показано на рис. 13-22. Провайдер службы RSVP (\Windows\System32\Rsvpsp.dll) передает QoS-запросы приложений

через RPC службе RSVP. Та в свою очередь контролирует сетевой трафик с помощью TC API. TC API, реализованный в \Windows\System32\Traffic.dll, посылает команды управления вводом-выводом драйверу GPC (Generic Packet Classifier) (\Windows\System32\Drivers\Msgpc.sys). Драйвер GPC — в тесном взаимодействии с планировщиком пакетов QoS (промежуточным драйвером NDIS) (\Windows\System32\Drivers\Psched.sys) — контролирует поток пакетов с компьютера в сеть, гарантируя уровни QoS, обещанные конкретным приложениям. При этом он вставляет в пакеты соответствующие заголовки QoS.

ПРИМЕЧАНИЕ В Windows XP и Windows Server 2003 служба RSVP по-прежнему работает, но действует лишь как посредник между приложениями и компонентами, управляющими трафиком.

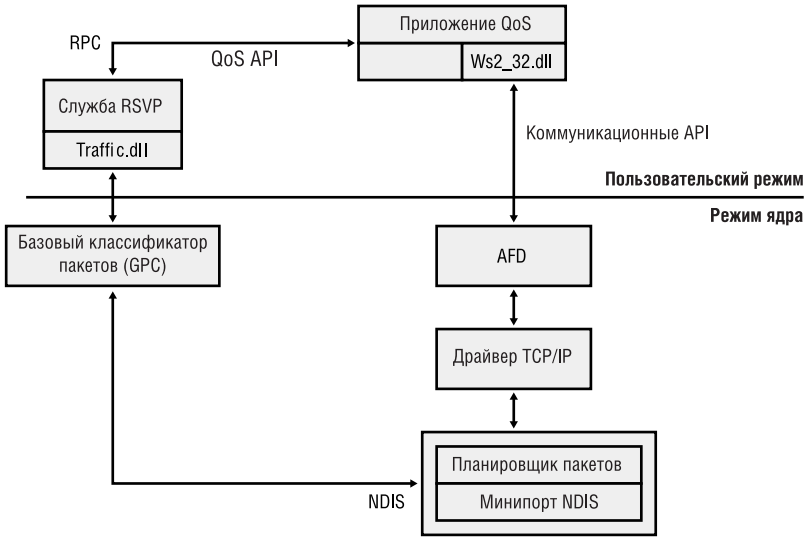


Рис. 13-22. *Архитектура QoS*

Привязка

Последний фрагмент головоломки под названием «сетевая архитектура Windows» — способ, посредством которого сетевые компоненты, расположенные на различных уровнях (сетевых API, драйверов транспортов TDI, драйверов NDIS), находят друг друга. Процесс соединения уровней называется *привязкой* (binding). Вы сами были свидетелем привязки, если хоть раз изменяли конфигурацию сети, добавляя или удаляя компоненты в окне свойств сетевого соединения.

Устанавливая сетевой компонент, вы должны предоставить его INF-файл (INF-файлы описаны в главе 9). Этот файл содержит инструкции, которым должны следовать API-функции установки, чтобы установить и сконфигурировать компонент, учитывая его зависимости от других компонентов. Раз-

работчик может указать зависимости для своего компонента, что позволит SCM загружать его в корректной очередности и только тогда, когда все компоненты, от которых он зависит, уже присутствуют в системе (о SCM см. главу 4). Привязки определяются механизмом привязки (bind engine) на основе информации из INF-файла компонента, что позволяют соединить его с другими компонентами, расположенными на различных уровнях. Эти соединения указывают, какие компоненты нижележащего уровня могут быть использованы сетевым компонентом данного уровня.

Так, служба рабочей станции (редиректор) автоматически привязывается к протоколам TCP/IP и NWLink. Порядок привязки, который можно увидеть на вкладке Adapters And Bindings (Адаптеры и привязки) диалогового окна Advanced Settings (Дополнительные параметры) (рис. 13-23), определяет приоритет привязки. (О том, как открыть это диалоговое окно, см. раздел «Поддержка нескольких редиректоров» ранее в этой главе.) Получив запрос на доступ к удаленному файлу, редиректор выдает запрос обоим драйверам протоколов. После того как редиректору приходит ответ, он дополнительно ждет ответы от любых драйверов протоколов с более высоким приоритетом. И только тогда редиректор возвращает результат вызывающей программе. Поэтому, присвоив высокий приоритет привязкам, которые дают максимальную производительность или применимы к большинству компьютеров в сети, можно добиться определенного выигрыша.

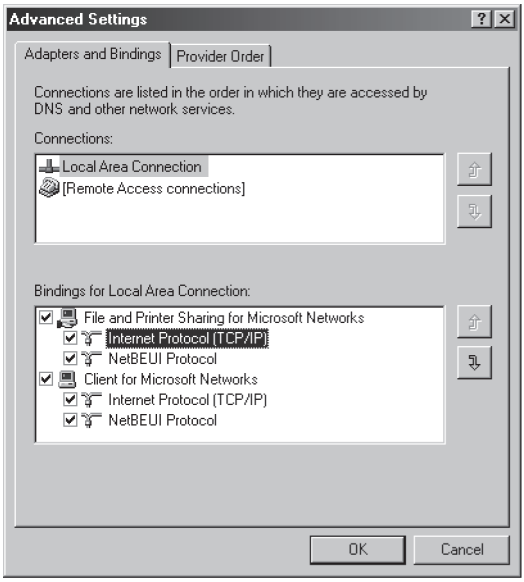


Рис. 13-23. *Редактирование привязок в диалоговом окне Advanced Settings*

Информация о привязках компонента содержится в параметре Bind подраздела Linkage того раздела реестра, в котором хранится конфигурация данного сетевого компонента. Например, информацию о привязках службы

рабочей станции можно найти в HKLM\SYSTEM\CurrentControlSet\Services\LanmanWorkstation\Linkage\Bind.

Многоуровневые сетевые сервисы

Windows включает сетевые сервисы, построенные на основе API и компонентов, представленных в этой главе. Описание возможностей и внутренних деталей их реализации выходит за рамки этой книги, но мы приведем здесь краткий обзор по удаленному доступу, службе каталогов Active Directory, Network Load Balancing, службе репликации файлов (File Replication Service, FRS) и распределенной файловой системе (Distributed File System, DFS).

Удаленный доступ

Windows Server с установленной службой Routing and Remote Access Service (Служба маршрутизации и удаленного доступа) позволяет клиентам удаленного доступа подключаться к серверам удаленного доступа и обращаться к таким сетевым ресурсам, как файлы, принтеры и службы, создавая иллюзию физического подключения к серверу удаленного доступа через локальную сеть. Windows поддерживает два типа удаленного доступа.

- **Удаленный доступ через телефонную линию (dial-up remote access)** Используется клиентом при подключении к серверу удаленного доступа через телефонную линию или иную телекоммуникационную инфраструктуру. Телекоммуникационная несущая среда используется для создания временного физического или виртуального соединения между клиентом и сервером.
- **Удаленный доступ через виртуальную частную сеть (virtual private network, VPN)** Позволяет клиентам VPN устанавливать с сервером виртуальное соединение типа «точка-точка» через IP-сеть, например Интернет.

Удаленный доступ отличается от удаленного управления, поскольку программное обеспечение удаленного доступа выступает в роли прокси-соединения с сетью Windows, тогда как программное обеспечение для удаленного управления выполняет приложения на сервере, предоставляя клиенту пользовательский интерфейс.

Active Directory

Active Directory — реализация службы каталогов LDAP (Lightweight Directory Access Protocol) в Windows. Active Directory основана на базе данных, в которой хранятся объекты, представляющие ресурсы, определенные приложениями в сети Windows. Например, в Active Directory хранится структура и список членов домена Windows, включая информацию об учетных записях и паролях пользователей.

Классы объектов и атрибуты, определяющие свойства объектов, задаются *схемой* (schema). Иерархическая организация объектов в схеме Active Directory напоминает логическую организацию реестра, где объекты-контейнеры могут хранить другие объекты, включая контейнеры.

Active Directory поддерживает несколько API, используемых клиентами для обращения к объектам в базе данных Active Directory.

- **LDAP C API** Предназначен для программ на C/C++ и использует сетевой протокол LDAP. Приложения, написанные на C/C++, могут работать с этим API напрямую, а приложения, написанные на других языках, — через транслирующие уровни.
- **ADSI (Active Directory Service Interfaces)** COM-интерфейс Active Directory, реализованный поверх LDAP и абстрагирующий от деталей программирования для LDAP. ADSI поддерживает несколько языков, в том числе Microsoft Visual Basic, C и Microsoft Visual C++. ADSI также доступен приложениям Windows Script Host (Сервер сценариев Windows).
- **MAPI (Messaging API)** Поддерживается для совместимости с клиентами Microsoft Exchange и клиентскими приложениями Outlook Address Book.
- **Security Account Manager (SAM) API** Базируется на сервисах Active Directory и предоставляет интерфейс пакетам аутентификации MSV1_0 (\Windows\System32\Msv1_0.dll) и Kerberos (\Windows\System32\Kdcsvc.dll).
- **Сетевые API Windows NT 4 (Net API)** Используются клиентами Windows NT 4 для доступа к Active Directory через SAM.
- **NTDS API** Применяется для просмотра SID и GUID в Active Directory (в основном через *DsCrackNames*), а также для управления каталогом и его репликацией. Несколько сторонних разработчиков написали приложения, позволяющие вести мониторинг Active Directory через этот API.

Active Directory реализована в виде файла базы данных (по умолчанию — в \Windows\Ntds\Ntds.dit), реплицируемой между контроллерами домена. Этой базой данных управляет служба каталогов Active Directory, которая является Windows-сервисом, выполняемым в процессе LSASS; при этом она использует DLL, реализующие структуру базы данных на диске и предоставляющие механизмы обновления на основе транзакций для поддержания целостности базы данных. В Windows 2000 хранилище базы данных Active Directory реализовано на основе ядра Extensible Storage Engine (ESE), применяемого в Microsoft Exchange Server 5.5, а в Windows 2003 Server — на основе ядра ESE, используемого в Microsoft Exchange Server 2000. Архитектура Active Directory показана на рис. 13-24.

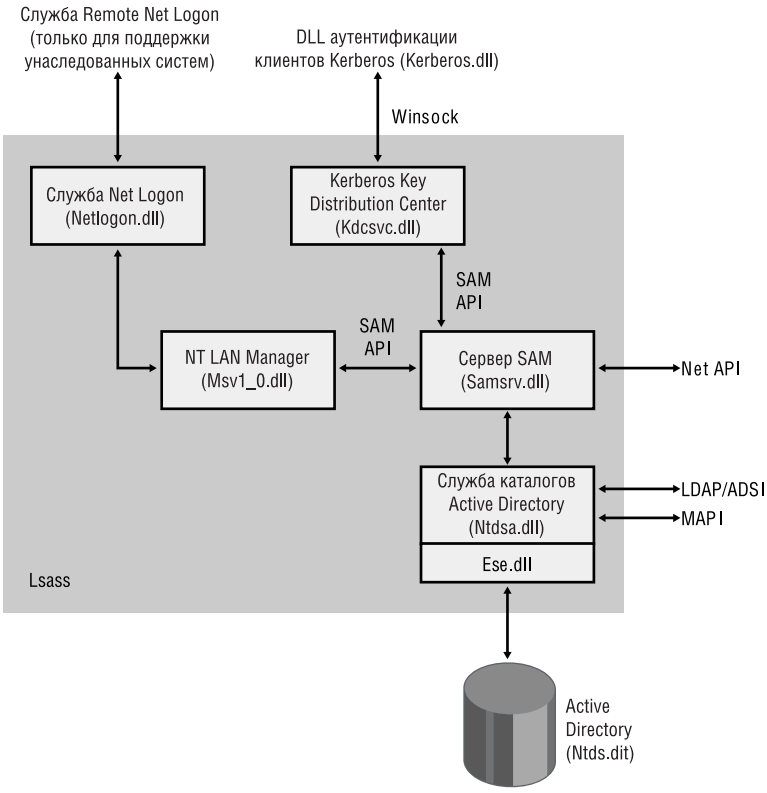


Рис. 13-24. *Архитектура Active Directory*

Network Load Balancing

Как мы уже говорили в этой главе, в основе компонента Network Load Balancing (Балансировка нагрузки сети), который входит в Windows Advanced Server, лежит технология промежуточных драйверов NDIS. Network Load Balancing допускает создание кластера, включающего до 32 компьютеров, которые в терминологии Network Load Balancing называются *узлами кластера* (cluster hosts). Кластер поддерживает единый виртуальный IP-адрес, публикуемый клиентам, и клиентские запросы поступают ко всем компьютерам кластера. Однако на запрос отвечает только один узел кластера. Драйверы NDIS компонента Network Load Balancing эффективно разделяют клиентское пространство между доступными узлами кластера по аналогии с распределенной обработкой. При таком подходе каждый узел обрабатывает свою порцию клиентских запросов, причем каждый клиентский запрос обрабатывается одним — и только одним — узлом. Если входящий в состав кластера узел определяет, что именно он должен обработать клиентский запрос, то этот узел позволяет запросу пройти до уровня драйвера TCP/IP и в конце концов достичь серверного приложения. Если на узле кластера про-

исходит авария, остальные узлы кластера распознают, что этот узел больше не способен обрабатывать запросы, и перераспределяют поступающие клиентские запросы между собой; при этом клиентские запросы отключенному узлу больше не посылаются. К кластеру можно подключить новый узел на замену потерпевшему аварию, и он автоматически примет участие в обработке клиентских запросов.

Network Load Balancing не является универсальным кластерным решением, поскольку серверные приложения, с которыми взаимодействуют клиенты, должны обладать определенными характеристиками. Во-первых, они должны поддерживать TCP/IP, а во вторых, уметь обрабатывать клиентские запросы на любой системе в кластере Network Load Balancing. Второе требование, как правило, означает, что приложения, у которых для обслуживания клиентских запросов должен быть доступ к общему состоянию (shared state), обязаны сами управлять этим состоянием. В Network Load Balancing не входят сервисы автоматического распределения общего состояния между узлами кластера. Приложения, идеально подходящие для Network Load Balancing, — Web-сервер со статичным информационным наполнением (контентом), Windows Media Server и Terminal Services (Службы терминалов). Пример работы Network Load Balancing показан на рис. 13-25.

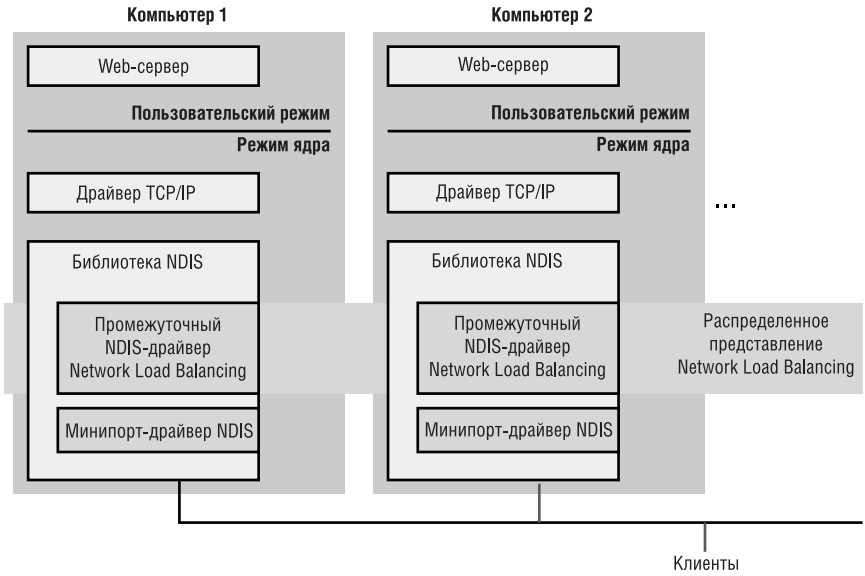


Рис. 13-25. *Так работает Network Load Balancing*

Служба репликации файлов

Служба репликации файлов (File Replication Service, FRS) входит в системы Windows Server. Она предназначена в основном для репликации содержимого каталога \SYSVOL контроллера домена (в этом месте контроллеры доме-

нов Windows хранят сценарии регистрации и групповые политики). Кроме того, FRS позволяет реплицировать общие ресурсы DFS (Distributed File System) между системами. FRS поддерживает *распределенную репликацию с несколькими хозяевами* (distributed multimaster replication), благодаря чему репликацию может проводить любой сервер. Когда реплицируемый файл или каталог изменяется, эти изменения распространяются на другие контроллеры домена.

Фундаментальное понятие в FRS — *набор репликации* (replica set), представляющий собой дерево каталогов, которое реплицируется между двумя или более системами по определенной топологии и расписанию, заданному администратором. Реплицированы могут быть только каталоги на томах NTFS, поскольку FRS использует журнал изменений NTFS для определения модификаций в файлах и каталогах, включенных в набор репликации. Поскольку FRS обеспечивает репликацию с несколькими хозяевами, теоретически она может поддерживать сотни и даже тысячи систем в наборе репликации, а топология соединения соответствующих компьютеров может быть совершенно произвольной (кольцо, звезда, сетка и др.). Кроме того, компьютеры могут участвовать в нескольких наборах репликации.

FRS реализована в виде Windows-сервиса (\Windows\System32\Ntfrs.exe), который использует аутентифицируемый RPC для взаимодействия со своими экземплярами, работающими на других компьютерах. Кроме того, поскольку Active Directory располагает собственными средствами репликации, FRS использует API-функции Active Directory для выборки конфигурационной информации из домена Active Directory.

DFS

DFS (Distributed File System) — сервис поверх службы рабочей станции, соединяющий отдельные файловые ресурсы в единое пространство имен. DFS обеспечивает клиентам прозрачный доступ к файловым ресурсам независимо от того, где находятся эти ресурсы — на локальном или удаленных компьютерах. Корнем пространства имен DFS должен быть файловый ресурс, определенный на компьютере с Windows Server.

В дополнение к унифицированному пространству имен сетевых ресурсов DFS дает и другие преимущества при использовании наборов репликации DFS. Администратор может создать набор репликации DFS минимум из двух сетевых ресурсов и использовать механизм репликации вроде FRS для копирования данных между ресурсами, входящими в набор репликации, и тем самым обеспечить синхронизацию их содержимого. DFS поддерживает несколько видов балансировки нагрузки, упорядочивая и/или выбирая сетевые ресурсы, входящие в набор репликации, при обращении клиента к данным из этого набора. DFS также обеспечивает высокую доступность данных, перенаправляя запросы на другие сетевые ресурсы из набора репликации, если какой-то из сетевых ресурсов временно недоступен.

Компоненты, образующие архитектуру DFS, показаны на рис. 13-26. Реализация DFS на серверной стороне включает Windows-сервис (\Windows\System32\Dfssvc.exe) и драйвер устройства (\Windows\System32\Drivers\Dfs.sys). Служба DFS отвечает за экспорт интерфейсов управления топологией DFS и поддержку топологии DFS либо в реестре (в отсутствие Active Directory), либо в Active Directory. Драйвер DFS принимает клиентский запрос и переадресует его системе, на которой находится запрошенный файл.

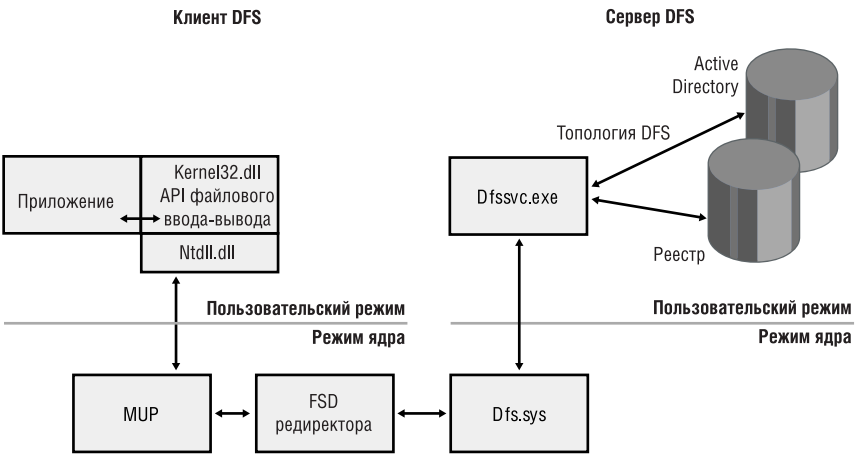


Рис. 13-26. *Компоненты DFS*

На клиентской стороне поддержка DFS реализована в драйвере MUP (о нем мы уже рассказывали) и использует редиректор CIFS для взаимодействия с серверами DFS на внутреннем уровне. Провайдер клиента DFS реализован в \Windows\System32\Ntlanman.dll. Когда клиент выдает запрос на ввод-вывод для файла в пространстве имен DFS, драйвер MUP на клиентской стороне взаимодействует с сервером, на котором находится этот файл, через подходящий редиректор.

Резюме

Сетевая архитектура Windows предоставляет гибкую инфраструктуру сетевым API, драйверам протоколов и сетевых адаптеров. Эта архитектура использует преимущества многоуровневого ввода-вывода, обеспечивая расширяемость сетевой поддержки по мере развития компьютерных сетей. При появлении нового протокола разработчики смогут создать транспорт TDI, реализующий этот протокол в Windows. Аналогичным образом новые API смогут взаимодействовать с существующими драйверами протоколов Windows. Наконец, набор сетевых API, реализованных в Windows, позволяет разработчикам сетевых приложений выбирать подходящие им реализации, поддерживающие разные модели программирования и протоколы.

ГЛАВА 14

Анализ аварийного дампа

Почти каждый пользователь Windows слышал о так называемом «синем экране смерти» (blue screen of death, BSOD) или даже видел его. Этим зловещим термином называют экран с синим фоном, показываемый при крахе или остановке Windows из-за катастрофического сбоя или внутренней ситуации, из-за которой стала невозможной дальнейшая работа системы.

В этой главе мы рассмотрим основные причины краха Windows, опишем информацию, выводимую на «синем экране» и расскажем о различных параметрах конфигурации, управляющих созданием *аварийного дампа (*crash dump) — копии системной памяти на момент краха, которая может помочь определить, какой именно компонент вызвал крах. В цели данного раздела *не* входит детальное рассмотрение способов выявления и устранения проблем с помощью анализа аварийного дампа Windows. Тем не менее в этом разделе показывается, как, проанализировав аварийный дамп, идентифицировать некорректно работающий драйвер или компонент. Для базового анализа аварийного дампа требуется минимум усилий и несколько минут времени. Анализ дампа стоит проводить, даже если проблемный драйвер удается выявить только с пятой или десятой попытки: успешно выполненный анализ позволит избежать потерь данных и простоя системы.

Почему происходит крах Windows?

Крах Windows (остановка системы и вывод «синего экрана») может быть вызван следующими причинами:

- необработанным исключением, вызванным драйвером устройства или системной функцией режима ядра, например из-за нарушения доступа к памяти (при попытке записи на страницу с атрибутом «только для чтения» или чтения по еще не спроецированному и, следовательно, недопустимому адресу);
- вызовом процедуры ядра, результатом которой является перераспределение процессорного времени из-за, например, ожидания на занятом объекте диспетчера ядра при IRQL уровня «DPC/dispatch» или выше (об IRQL см. главу 3);
- обращением к данным на выгруженной из памяти странице при IRQL уровня «DPC/dispatch» или выше (что требует от диспетчера памяти ждать

операции ввода-вывода, а это, как уже говорилось, невозможно на таких уровнях IRQL, поскольку требует перераспределения процессорного времени);

- явным вызовом краха системы драйвером устройства или системной функцией (через функцию *KeBugCheckEx*) при обнаружении поврежденных внутренних данных или в ситуации, когда продолжение работы системы грозит таким повреждением;
- аппаратной ошибкой, например ошибкой аппаратного контроля или появлением немаскируемого прерывания (Non-Maskable Interrupt, NMI).

В Microsoft проанализировали аварийные дампы, отправляемые пользователями Windows XP на сайт Microsoft Online Crash Analysis (OCA) (о нем еще пойдет речь в этой главе), и обнаружили, что причины краха систем распределяются, как показано на диаграмме на рис. 14-1 (по состоянию на апрель 2004 года).

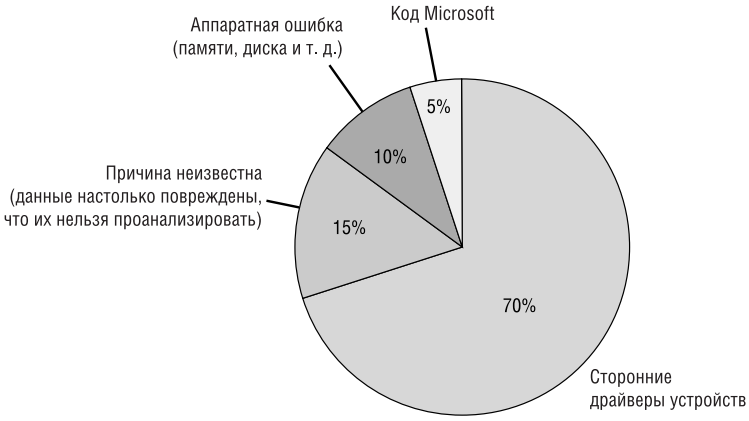


Рис. 14-1. *Причины краха систем по данным OCA*

Когда драйвер устройства или компонент режима ядра вызывает необрабатываемое исключение, перед Windows встает трудная дилемма. Какая-то часть операционной системы, имеющая право доступа к любым аппаратным устройствам и любому участку памяти, сделала нечто такое, чего делать нельзя.

Но почему при этом обязательно должен произойти крах Windows? Почему бы не проигнорировать это исключение и не позволить драйверам работать дальше, как ни в чем не бывало? Ведь не исключено, что ошибка носила локальный характер и соответствующий компонент как-нибудь сумеет после нее восстановиться. Но гораздо вероятнее, что обнаруженное исключение связано с более серьезными проблемами, например с повреждением памяти или со сбоями в работе оборудования. Тогда дальнейшее функционирование системы скорее всего приведет к еще большему числу исключений и порче данных на дисках и других периферийных устройствах, а это слишком рискованно.

«Синий экран»

Независимо от причины реальный крах системы вызывается функцией *KeBugCheckEx* (документирована в Windows DDK). Она принимает так называемый *стоп-код* (stop code), или *контрольный код ошибки* (bug check code), и четыре параметра, интерпретируемые с учетом стоп-кода. *KeBugCheckEx* маскирует все прерывания на всех процессорах системы, а затем переключает видеоадаптер в графический режим VGA с низким разрешением (поддерживаемый всеми видеокартами, совместимыми с Windows) и выводит на синем фоне значение стоп-кода и несколько строк текста с рекомендациями относительно дальнейших действий. Наконец, *KeBugCheckEx* вызывает все зарегистрированные (с помощью функции *KeRegisterBugCheckCallback*) функции обратного вызова драйверов устройств при ошибке (device driver bug check callbacks), чтобы они могли остановить свои устройства. (Системные структуры данных могут быть настолько серьезно повреждены, что «синий экран» может и не появиться.) Образец «синего экрана» Windows XP показан на рис. 14-2.

ПРИМЕЧАНИЕ В Windows XP Service Pack 1 (или выше) и в Windows Server 2003 введена функция *KeRegisterBugCheckReasonCallback*, позволяющая драйверам устройств добавить данные в аварийный дамп или вывести информацию аварийного дампа на альтернативное устройство.

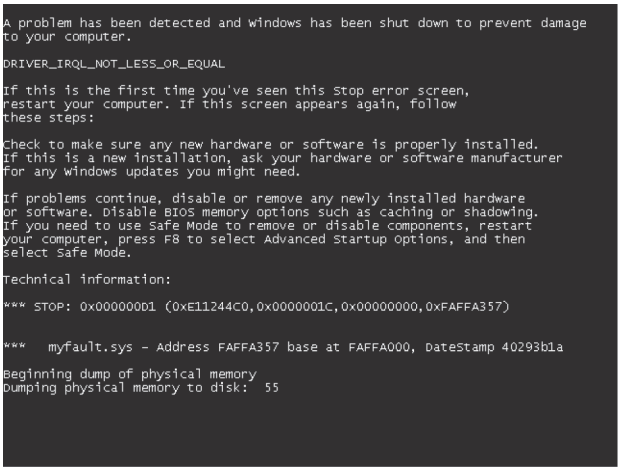


Рис. 14-2. *Пример «синего экрана»*

В Windows 2000 KeBugCheckEx выводит текстовое представление стоп-кода, его числовое значение и четыре параметра вверху «синего экрана», но в Windows XP и Windows Server 2003 числовое значение и параметры показываются внизу «синего экрана».

В первой строке выводится стоп-код и значения четырех дополнительных параметров, переданных в *KeBugCheckEx*. Строка вверху экрана пред-

ставляет собой текстовый эквивалент числового идентификатора стоп-кода. В примере на рис. 14-2 стоп-код 0x000000D1 соответствует IRQL_NOT_ LESS_OR_ EQUAL. Если параметр содержит адрес части операционной системы или кода драйвера устройства (как на рис. 14-2), Windows выводит базовый адрес соответствующего модуля, дату и имя файла драйвера. Одной этой информации может оказаться достаточно для идентификации сбойного компонента.

Хотя стоп-кодов более сотни, большинство из них очень редко или вообще никогда не встречается в рабочих системах. Причины краха Windows могут быть представлены довольно небольшой группой стоп-кодов. Кроме того, не забывайте, что смысл дополнительных параметров зависит от конкретного стоп-кода (но не для всех стоп-кодов предусматривается расширенная информация, передаваемая через эти параметры). Тем не менее, анализ стоп-кода и значений параметров (если таковые есть) может, по крайней мере, помочь в выявлении сбойного компонента (или аппаратного устройства, вызывающего крах).

Информацию, необходимую для интерпретации стоп-кодов, можно найти в разделе «Bug Checks (Blue Screens)» справочного файла Windows Debugging Tools. (Сведения о Windows Debugging Tools см. в главе 1.) Кроме того, можно поискать стоп-код и имя проблемного устройства или приложения в Microsoft Knowledge Base (*http://support.microsoft.com*). В ней можно найти информацию о способах исправления ошибки, об обновлениях или сервисных пакетах, решающих проблему, с которой вы столкнулись. Файл Bugcodes.h в Windows DDK содержит полный список из примерно 150 стоп-кодов с детальным описанием некоторых из них.

«Синие экраны» часто возникают после установки нового программного обеспечения или оборудования. Если вы видите «синий экран» сразу после установки нового драйвера на раннем этапе перезагрузки, то можете вернуть прежнюю конфигурацию системы, нажав клавишу F8 и выбрав из дополнительного загрузочного меню команду Last Known Good Configuration (Последняя удачная конфигурация). Тогда Windows использует копию раздела реестра, где были зарегистрированы драйверы устройств (HKLM\SYSTEM\ CurrentControlSet\Services) при последней успешной загрузке (до установки нового драйвера). Последней удачной конфигурацией считается последняя конфигурация, в которой успешно завершилась загрузка всех сервисов и драйверов и был выполнен минимум один успешный вход в систему. (О последней удачной конфигурации более подробно рассказывается в главе 5.)

Если это не помогает и вы по-прежнему видите «синие экраны», то самый очевидный подход — удалить компоненты, установленные перед появлением первого «синего экрана». Если после установки уже прошло некоторое время или вы одновременно добавили несколько устройств либо драйверов, обратите внимание на имена драйверов, указываемые в каких-либо параметрах на «синем экране». Если там есть ссылка на недавно установленные компоненты (например, Scsiport.sys в случае установки нового SCSI-диска), причина сбоя скорее всего связана именно с ними.

Имена многих драйверов весьма загадочны, но вы можете выяснить, какие устройства или программные компоненты соответствуют данному имени. Для этого просмотрите раздел реестра HKLM\SYSTEM\CurrentControlSet\Services, где Windows хранит регистрационную информацию для каждого драйвера в системе, и попробуйте найти имя сервиса и сопоставленный с ним драйвер устройства. Описание найденного драйвера содержится в параметрах DisplayName и Description, здесь также описывается предназначение некоторых драйверов. Так, строка «Virus Scanner», обнаруженная в DisplayName, говорит о том, что драйвер является частью антивирусной программы. Список драйверов также можно вывести с помощью утилиты System Information (Сведения о системе): раскройте в ней узел Software Environment (Программная среда) и выберите System Drivers (Системные драйверы).

Однако чаще всего информации, сообщаемой стоп-кодом и сопоставленными с ним параметрами, недостаточно для устранения сбоя, приводящего к краху системы. Так, чтобы выяснить точное имя драйвера или системного компонента, вызывающего крах, может понадобиться анализ стека вызовов режима ядра. Поскольку в Windows после краха системы по умолчанию следует перезагрузка и у вас вряд ли будет время для изучения информации, представленной на «синем экране», Windows пытается записывать информацию о крахе системы на диск для последующего анализа. Эта информация помещается в файлы аварийного дампа.

Файлы аварийного дампа

По умолчанию все Windows-системы настраиваются на запись информации о состоянии системы на момент краха. Соответствующие настройки можно увидеть так: откройте System (Система) в Control Panel (Панель управления), в окне свойств системы перейдите на вкладку Advanced (Дополнительно) и щелкните кнопку Startup And Recovery (Загрузка и восстановление). На рис. 14-3 показаны настройки по умолчанию для системы Windows XP Professional.

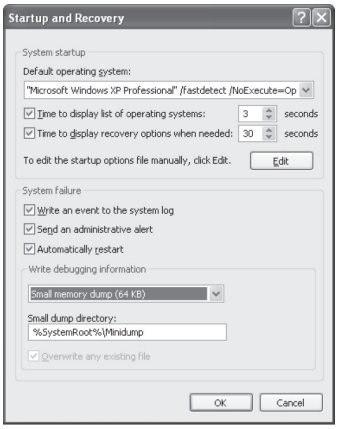


Рис. 14-3. *Параметры аварийного дампа*

При крахе системы может быть зарегистрировано три уровня информации.

- **Complete memory dump (Полный дамп памяти)** Полный дамп памяти представляет собой все содержимое физической памяти на момент краха. Для такого дампа нужно, чтобы размер страничного файла был равен, как минимум, объему физической памяти плюс 1 Мб (для заголовка). Этот параметр используется реже всего, так как в системах с большим объемом памяти страничный файл будет слишком велик. Windows NT 4 поддерживает только этот тип файлов аварийного дампа. Кроме того, этот параметр используется по умолчанию в системах Windows Server.
- **Kernel memory dump (Дамп памяти ядра)** Этот вариант дампа включает лишь страницы (как для чтения, так и для записи) режима ядра, находящиеся в физической памяти на момент краха. Страницы, принадлежащие пользовательским процессам, не включаются. Поскольку только код режима ядра может напрямую вызывать крах Windows, содержимое страниц пользовательских процессов обычно ничего не дает для понимания причин краха. Кроме того, все структуры данных, используемые при анализе аварийного дампа, — список выполняемых процессов, стек текущего потока и список загруженных драйверов — хранятся в неподкачиваемой памяти, содержимое которой запоминается в дампе памяти ядра. Заранее предсказать объем дампа памяти ядра нельзя, поскольку он зависит от объема памяти ядра, выделенной операционной системой и драйверами.
- **Small memory dump (Малый дамп памяти)** Размер этого дампа (вариант по умолчанию в системах Windows Professional) составляет 64 Кб (128 Кб в 64-битных системах). Такой дамп еще называют *минидампом* (minidump) или *минимальным дампом* (triage dump). Он включает в себя стоп-код с параметрами, список загруженных драйверов устройств, структуры данных, описывающие текущие процесс и поток (EPROCESS и ETHREAD, которые рассматриваются в главе 6), а также стек ядра для вызвавшего крах потока.

Полный дамп памяти является надмножеством двух других дампов, но у него есть недостаток: его размер зависит от объема физической памяти системы и, следовательно, он может оказаться слишком большим. Мощные серверные системы, оснащенные несколькими гигабайтами памяти, — не такая уж редкость. Записываемые на них файлы полного аварийного дампа будут слишком велики для закачивания на FTP-сервер или прожигания на CD. Поскольку в большинстве случаев код и данные пользовательского режима не используются при анализе аварийных дампов (ведь причиной краха являются проблемы, связанные с памятью ядра, системные структуры данных также содержатся в памяти ядра), большая часть данных, сохраненных в полном дампе памяти, не нужна для анализа и впустую увеличивает размер файла дампа. Наконец, еще один недостаток в том, что размер страничного файла на загрузочном томе (содержащем каталог \Windows) должен быть равен объему физической памяти системы плюс 1 Мб. Поскольку необходимость в страничном файле, как правило, уменьшается с ростом объема физической памяти, это требование означает, что страничный файл будет неоправ-

данно большим. Поэтому приходится признать, что лучше использовать малый дамп памяти или дамп памяти ядра.

Преимущество минидампа — его небольшой размер, благодаря которому, например, удобно передавать дамп по электронной почте. При каждом крахе в каталог \Windows\Minidump записывается файл с уникальным именем, начинающимся со строки «Mini», за которой идут дата и порядковый номер (например, Mini082604-01.dmp). Недостаток минидампов в том, что для их анализа нужны именно те образы, которые использовались системой, сгенерировавшей дамп. (Даже для самого простого анализа, как минимум, необходима копия соответствующего Ntoskrnl.exe.) Это может стать проблемой, если вы анализируете дамп не на той системе, где он был создан. Однако на сервере символов Microsoft есть образы (и символы) для систем Windows XP и более поздних версий, поэтому можно задать в отладчике путь к образу, указывающий на сервер символов, и отладчик автоматически скачает нужные образы. (Конечно, на сервере символов Microsoft нет образов устанавливаемых вами драйверов сторонних производителей.)

Более существенный недостаток — такой дамп содержит ограниченное количество данных, что может помешать эффективному анализу. С минидампами можно работать, даже если вы настроили систему на генерацию дампа памяти ядра или полного дампа, — просто откройте более объемный дамп в Windbg и извлеките минидамп командой *.dump /m*. Заметьте: в Windows XP и Windows Server 2003 минидамп автоматически создается, даже если система настроена на генерацию полного дампа памяти или дампа памяти ядра.

ПРИМЕЧАНИЕ Выполнив команду *.dump* в Livekd, можно сгенерировать образ памяти работающей системы, чтобы, не останавливая систему, получить дамп для анализа в автономном режиме. Такой подход полезен, когда в системе проявляются какие-то проблемы, но она продолжает обслуживать клиентов и вы хотели бы устранить проблемы, не прерывая обслуживание. Полученный в результате дамп не обязательно будет полностью корректным, так как содержимое различных областей памяти извлекается в разные моменты времени, но может содержать информацию, полезную для анализа.

Золотой серединой является дамп памяти ядра. Он содержит всю физическую память режима ядра, и, следовательно, позволяет вести анализ на том же уровне, что и полный дамп памяти, но не содержит код и данные пользовательского режима, обычно не относящиеся к проблеме, и поэтому имеет значительно меньший размер. Так, в системе с 256 Мб памяти под управлением Windows XP дамп памяти ядра занимает 34 Мб, а в системе с Windows XP и 1,5 Гб памяти этот дамп требует 72 Мб.

Когда вы настраиваете параметры дампа памяти ядра, система проверяет, достаточен ли размер страничного файла (в соответствии с таблицей 14-1), но это всего лишь оценочные размеры, поскольку предсказать размер дампа памяти ядра невозможно. Причина, по которой невозможно заранее определить размер дампа памяти ядра, состоит в том, что этот размер зави-

сит от количества памяти режима ядра, используемой операционной системой и драйверами, выполнявшимися на компьютере в момент краха.

Таблица 14-1. *Минимальные размеры страничного файла для дампов ядра*

Объем системной памяти	Минимальный размер страничного файла для генерации дампов ядра (Мб)
< 128 Мб	50
< 4 Гб	200
< 8 Гб	400
>= 8 Гб	800

Таким образом, может оказаться, что в момент краха системы страничный файл будет слишком мал для того, чтобы вместить дамп ядра. Если вы хотите узнать размер дампа ядра для своей системы, вызовите крах системы вручную: сконфигурируйте систему так, чтобы можно было вручную вызывать ее крах с консоли, или воспользуйтесь программой Notmyfault. (В этой главе описаны оба подхода.) После перезагрузки вы сможете проверить, сгенерирован ли дамп памяти ядра, и по его размеру оценить, каким должен быть размер страничного файла для вашего загрузочного тома. Для единообразия можно задавать для 32-разрядных систем размер страничного файла 2 Гб плюс 1 Мб, поскольку 2 Гб — максимальный размер адресного пространства режима ядра.

Наконец, даже если система в случае краха успешно записывает аварийный дамп в страничный файл, нужно, чтобы на диске хватало места для извлечения файла дампа. Если места не хватит, аварийный дамп пропадет, поскольку используемое им пространство страничного файла высвободится и будет перезаписано, когда система начнет использовать страничный файл. Если на загрузочном томе недостаточно места для сохранения файла memory.dmp, можно задать путь на другом жестком диске в диалоговом окне, показанном на рис. 14-3.

Генерация аварийного дампа

При загрузке система получает параметры аварийного дампа из раздела реестра HKLM\System\CurrentControlSet\Control\CrashControl. Если задана генерация дампа, система создает копию минипорт-драйвера диска (disk miniport driver), используемую для записи загрузочного тома в память и присваивает ей то же имя, что и у минипорта, но с префиксом «dump_». Кроме того, система подсчитывает и сохраняет контрольную сумму для компонентов, используемых при записи аварийного дампа: скопированного минипорт-драйвера диска, функций диспетчера ввода-вывода, записывающих дамп, и карты области, в которой располагается страничный файл на загрузочном томе. Когда вызывается функция *KeBugCheckEx*, она заново пересчитывает контрольную сумму и сравнивает новую контрольную сумму с полученной при загрузке. Если они не совпадают, функция не записывает аварийный дамп, так как это может привести к сбою диска или повреждению данных на

диске. Если контрольные суммы совпали, *KeBugCheckEx* записывает информацию дампа прямо в секторы диска, занимаемые страничным файлом, минуя драйвер файловой системы (который, возможно, поврежден или даже является причиной краха).

Когда SMSS в процессе загрузки активизирует постраничную подкачку, система проверяет, не содержится ли в страничном файле на загрузочном томе аварийный дамп, и защищает ту часть страничного файла, которая отведена под дамп. В результате на раннем этапе загрузки часть страничного файла или весь этот файл выводится из использования, что может вызвать системные уведомления о нехватке виртуальной памяти, однако это лишь временное явление. При дальнейшей загрузке Winlogon определяет, содержится ли дамп в страничном файле, вызывая недокументированную API-функцию *NtQuerySystemInformation*. Если дамп есть, запускается процесс Savedump (\Windows\System32\Savedump.exe), который извлекает аварийный дамп из страничного файла и записывает его в заданное место. Эти операции показаны на рис. 14-4.

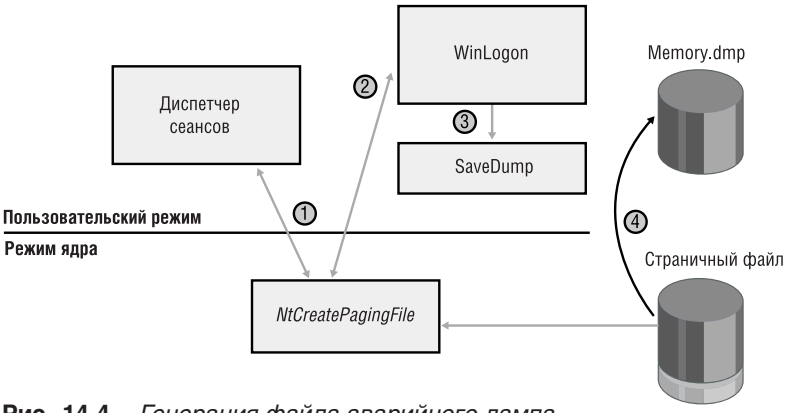


Рис. 14-4. *Генерация файла аварийного дампа*

Windows Error Reporting

Как уже говорилось в главе 3, в Windows XP и Windows Server 2003 имеется механизм Windows Error Reporting, позволяющий автоматически передавать данные о сбоях процессов и системы на анализ в Microsoft (или на внутренний сервер отчетов об ошибках). По умолчанию этот механизм включен. На его работу можно повлиять, изменив поведение процесса Savedump, который выполняет следующую дополнительную операцию: при перезагрузке после краха проверяет, настроена ли система на отправку аварийного дампа на анализ в Microsoft (или на закрытый сервер). На рис. 14-5 показано диалоговое окно Error Reporting (Отчет об ошибках), которое можно открыть с вкладки Advanced (Дополнительно) апплета System (Система) панели управления. В этом диалоговом окне можно настроить параметры системных отчетов об ошибках, хранящиеся в разделе реестра HKLM\Software\Microsoft\PCHealth\ErrorReporting.

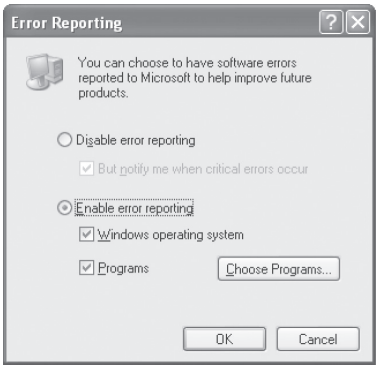


Рис. 14-5. *Диалоговое окно настройки Error Reporting*

После перезагрузки, вызванной крахом, Savedump проверяет несколько параметров, содержащихся в разделе ErrorReporting: Showui, DoReport и IncludeKernelFaults. Если все они имеют значение true, Savedump выполняет следующие операции по подготовке отчета о крахе системы к отправке на сайт Microsoft Online Crash Analysis (OCA) (или на внутренний сервер отчетов об ошибках, если это задано в настройках).

1. Если сгенерированный дамп не является минидампом, извлекает из файла дампа минидамп и записывает его в каталог по умолчанию — \Windows\Minidumps.
2. Записывает имя файла минидампа в HKLM\Software\Microsoft\PCHealth\ErrorReporting\KernelFaults.
3. Добавляет команду запуска утилиты Dumprep (\Windows\System32\Dumprep.exe) в раздел HKLM\Software\Microsoft\Windows\CurrentVersion\Run, чтобы Dumprep запустилась при первом входе пользователя в систему.

Анализ аварийных дампов через Интернет

Когда запускается утилита Dumprep (в результате того, что Savedump добавила в реестр соответствующее значение), эта утилита проверяет те же три параметра, что и Savedump, чтобы определить, должна ли система отправить отчет об ошибке после перезагрузки, вызванной крахом. Если должна, Dumprep генерирует XML-файл, содержащий базовое описание системы, в том числе версию операционной системы, список драйверов, установленных на компьютере, и список драйверов Plug and Play, загруженных в момент краха. Затем Dumprep выводит диалоговое окно, показанное на рис. 14-6, запрашивая у пользователя, нужно ли отправить в Microsoft отчет об ошибке. Если пользователь указал, что нужно, и это не противоречит групповым политикам, Dumprep отправляет XML-файл и минидамп на сайт *http://watson.microsoft.com*, который пересылает данные на серверную ферму, где отчеты автоматически анализируются (об этом см. следующий раздел). Через групповые политики администраторы могут настроить свои системы так, чтобы данные об ошибках направлялись во внутренний сетевой каталог,

предназначенный для сбора данных об ошибках. В дальнейшем эти данные можно обрабатывать с помощью Microsoft Corporate Error Reporting (CER) Toolkit, доступного только избранным клиентам Microsoft Software Assurance (информацию см. по ссылке *http://www.microsoft.com/resources/satech/cer*).

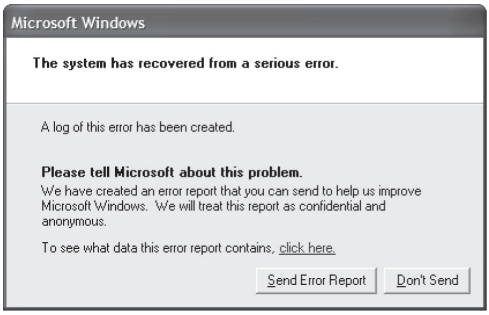


Рис. 14-6. *Диалоговое окно, предлагающее отправить отчет об ошибке*

Ферма серверов автоматического анализа использует тот же механизм, что и разработанные Microsoft отладчики ядра, в которые вы можете загрузить аварийный дамп (вскоре мы о них расскажем). При анализе генерируется так называемый *идентификатор типа* (bucket ID) — сигнатура, идентифицирующая определенный тип краха. Ферма серверов выполняет запрос к базе данных, пытаясь по идентификатору типа найти решение проблемы, вызвавшей крах, и отправляет утилите Dumprep URL со ссылкой на сайт OCA (*http://oca.microsoft.com*). Dumprep запускает Web-браузер, чтобы открыть страницу сайта OCA с предварительными результатами анализа дампа. Если решение проблемы найдено, на странице выводятся инструкции о том, где получить критическое исправление, сервисный пакет или обновление стороннего драйвера; в ином случае предоставляется возможность получать информацию о ходе анализе дампа по электронной почте.

Если у организации нет доступа к Интернету или она не собирается автоматически отправлять аварийные дампы в Microsoft, то через групповые политики можно указать, что данные об ошибках должны храниться во внутреннем сетевом каталоге; в дальнейшем их можно будет обрабатывать с помощью Microsoft CER Toolkit, упоминавшегося выше.

Базовый анализ аварийных дампов

Если при анализе, выполненном OCA, не удалось найти решение проблемы или если вы не сумели отправить аварийный дамп на сайт OCA (например, если этот дамп сгенерирован Windows 2000, не поддерживающей OCA), то вы можете самостоятельно проанализировать дамп. Как уже говорилось, когда вы загружаете аварийный дамп в Windbg или Kd, эти отладчики ядра применяют тот же механизм анализа, что и OCA. Иногда даже базового анализа достаточно для выявления проблемы. Таким образом, если вам повезет, вы найдете решение проблемы путем автоматического анализа аварийного дампа. Но даже если и не повезет, существуют простые методики выявления причин краха.

В этом разделе поясняется, как выполнить базовый анализ аварийного дампа, затем даются рекомендации, как с помощью Driver Verifier (с которым вы познакомились в главе 7) перехватывать операции некорректно написанных драйверов, приводящие к повреждению системы, и получать аварийные дампы, анализ которых может выявить проблему.

Notmyfault

Различные виды краха системы, рассматриваемые здесь, можно вызвать с помощью утилиты Notmyfault (*wwwsysinternals.com/windowsinternals*). Notmyfault состоит из исполняемого файла Notmyfault.exe и драйвера Myfault.sys. Когда вы запускаете исполняемый файл Notmyfault, он загружает драйвер и выводит диалоговое окно, показанное на рис. 14-7. В этом окне вы можете выбрать различные варианты краха системы или указать, что драйвер должен вызвать утечку памяти из пула подкачиваемой памяти. Доступны наиболее распространенные (по статистике Microsoft Product Support Services) виды краха системы. После того как вы выбрали параметр и щелкнули кнопку Do Bug, исполняемый файл через API-функцию *DeviceIoControl* обращается к драйверу и указывает ему, ошибка какого типа должна произойти. Заметьте: лучше экспериментировать, вызывая крах системы через Notmyfault, на тестовой или виртуальной системе, так как полностью исключить вероятность того, что поврежденная память не будет записана на диск, нельзя.

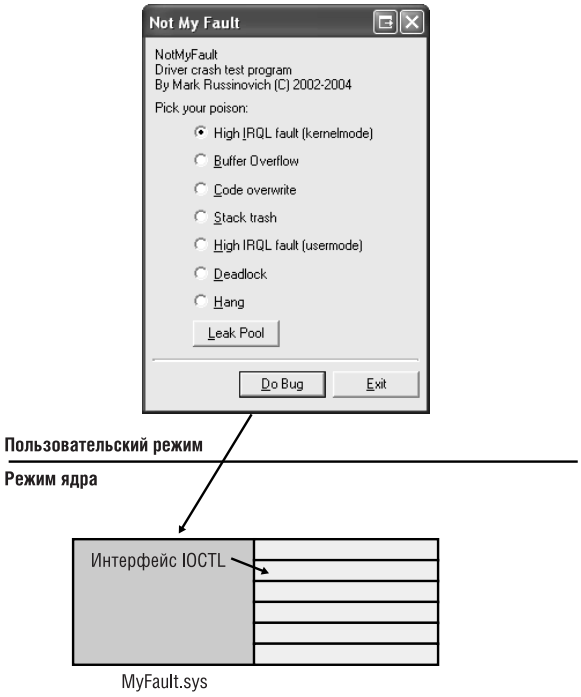


Рис. 14-7. *Notmyfault*

ПРИМЕЧАНИЕ Имена исполняемого файла и драйвера Notmyfault («не моя вина») отражают тот факт, что приложение, выполняемое в пользовательском режиме, не может напрямую вызвать крах системы. Исполняемый файл Notmyfault способен вызвать крах системы, только загрузив драйвер, который выполнит запрещенную операцию в режиме ядра.

Базовый анализ

Самый простой для отладки крах вызывается при выборе переключателя High IRQL Fault (Kernelmode) и нажатии кнопки Do Bug. Тогда драйвер выделит страницу в пуле подкачиваемой памяти, освободит страницу, поднимет уровень IRQL выше «DPC/dispatch», а затем обратится к освобожденной странице. (Об IRQL см. главу 3.) Если это не приведет к краху, система продолжит считывать память после конца страницы до тех пор, пока не произойдет крах из-за обращения к недействительной странице. Таким образом, драйвер выполняет несколько недопустимых операций.

1. Ссылается на память, которая ему не принадлежит.
2. Обращается к пулу подкачиваемой памяти при IRQL уровня «DPC/dispatch» или выше, что недопустимо, так как при таких IRQL ошибки страниц не разрешены.
3. Выходит за конец выделенной области памяти и пытается обратиться к памяти, которая потенциально может быть недействительной.

Первое обращение к странице не обязательно должно вызвать крах, если страница, освобожденная драйвером, остается в системном рабочем наборе. (О системном рабочем наборе см. главу 7.)

Загрузив в Kd аварийный дамп, сгенерированный при таком крахе, вы увидите следующие результаты:

```
Microsoft (R) Windows Debugger  Version 6.3.0011.2
Copyright (c) Microsoft Corporation. All rights reserved.

Loading Dump File [c:\windows\memory.dmp]
Kernel Summary Dump File: Only kernel address space is available

Symbol search path is: srv*c:\symbols*http://msdl.microsoft.com/
download/symbols

Executable search path is:
Windows 2000 Kernel Version 2195 UP Free x86 compatible
Product: LanManNt, suite: Enterprise TerminalServer
Kernel base = 0x80400000 PsLoadedModuleList = 0x8046a4c0
Debug session time: Mon Apr 05 18:28:44 2004
System Uptime: 0 days 0:09:20.105
Loading Kernel Symbols
```

```
.....................................................................................
Loading unloaded module list
No unloaded module list present
Loading User Symbols
..........
*******************************************************************
*                                                                 *
*                        Bugcheck Analysis                        *
*                                                                 *
*******************************************************************

Use !analyze -v to get detailed debugging information.

BugCheck D1, {bead1800, 1c, 0, ec1fb357}

*** ERROR: Module load completed but symbols could not be loaded
 for myfault.sys

*** WARNING: Unable to verify checksum for NotMyfault.exe
*** ERROR: Module load completed but symbols could not be loaded
 for NotMyfault.exe
Probably caused by : myfault.sys ( myfault+357 )

Followup: MachineOwner
---------

kd>
```

Прежде всего следует заметить, что Kd сообщает об ошибках при загрузке символов для Myfault.sys и Notmyfault.exe. Этого можно было ожидать, поскольку файлы символов для них нельзя обнаружить по пути поиска файлов символов (который указывает на сервер символов Microsoft). Вы будете получать аналогичные ошибки для драйверов сторонних производителей и исполняемых файлов, не входящих в операционную систему.

Текст, содержащий результаты анализа, достаточно краток: показаны числовой стоп-код и контрольные параметры, далее идет строка «probably caused by». В ней указан драйвер, который, с точки зрения механизма анализа, является наиболее вероятной причиной ошибки. В данном случае наш драйвер попал на заметку, и эта строка указывает прямо на Myfault.sys, поэтому проводить анализ вручную нет нужды.

Строка «Followup», как правило, не несет полезной информации — эти данные используются в Microsoft, когда отладчик ищет имя модуля в файле Triage.ini, содержащемся в подкаталоге Triage установочного каталога Debugging Tools for Windows. В версии этого файла, используемой внутри Microsoft, перечислены разработчики или группы, которые должны анализировать крах системы, вызываемый тем или иным драйвером, и, если удалось найти разработчика или группу, соответствующее имя выводится в строке Followup.

Детальный анализ

Во всех случаях, даже когда удалось выявить сбойный драйвер с помощью базового анализа аварийного дампа Notmyfault, нужно проводить детальный анализ командой:

```
!analyze -v
```

Первое очевидное отличие детального анализа и анализа по умолчанию состоит в том, что в первом случае выводится описание стоп-кода и его параметров. Ниже приведен вывод этой команды для того же дампа:

```
DRIVER_IRQL_NOT_LESS_OR_EQUAL (d1)
An attempt was made to access a pageable (or completely invalid) address at an
interrupt request level (IRQL) that is too high.  This is usually caused by
drivers using improper addresses.
If kernel debugger is available get stack backtrace.
Arguments:
Arg1: bead1800, memory referenced
Arg2: 0000001c, IRQL
Arg3: 00000000, value 0 = read operation, 1 = write operation
Arg4: ec1fb357, address which referenced memory
```

Таким образом, вам не придется открывать справочный файл, чтобы получить ту же информацию. Иногда выводимый текст содержит рекомендации по устранению неполадок — вы увидите такой пример в следующем разделе, где рассматривается углубленный анализ дампов.

Другая потенциально полезная информация, выводимая при детальном анализе — трассировочные данные стека потока, выполнявшегося в момент краха. Вот как она выглядит для того же дампа:

```
STACK_TEXT:
WARNING: Stack unwind information not available.
Following frames may be wrong.
b7e6dc34 804ac5de ff7c4040 beb4bf68 ff80a9c8 myfault+0x357
b7e6dd00 804a8f1e 00000070 00000000 00000000
 nt!IopXxxControlFile+0x5e4
b7e6dd34 80461691 00000070 00000000 00000000
 nt!NtDeviceIoControlFile+0x28
b7e6dd34 77f96be2 00000070 00000000 00000000
 nt!KiSystemService+0xc4
0012f4a0 77e84c9b 00000070 00000000 00000000
 ntdll!ZwDeviceIoControlFile+0xb
0012f504 004017c3 00000070 83360018 00000000
 KERNEL32!DeviceIoControl+0x100
000200ac 00000000 00000000 00000000 00000000
 NotMyfault+0x17c3
```

Приведенный выше стек показывает, что образ исполняемого файла Notmyfaul, показанный внизу, вызывал функцию *DeviceIoControl* в Kernel32.dll,

которая в свою очередь вызвала *ZwDeviceIoControlFile* в Ntdl.dll, и т. д., пока система, наконец, не рухнула при выполнении инструкции в образе Myfault. Стеки вызовов, подобные этому, могут оказаться полезными, поскольку иногда причиной краха системы является то, что один драйвер передал другому неправильно отформатированные, поврежденные или недопустимые параметры. Драйвер, передавший некорректные данные, способные вызвать крах системы, можно выявить при анализе, просмотрев стек вызовов, из которого видно, что было обращение к другому драйверу. В данном простом примере в стеке вызовов показан только драйвер myfault. (Модуль «nt» — это Ntoskrnl.)

Если вам не известен драйвер, выявленный при анализе, выполните команду *lm* (аббревиатура от «list modules»), чтобы посмотреть информацию о версии драйвера. Укажите параметры *k* (kernel modules), *v* (verbose), *m* (match), а затем имя драйвера и символ подстановки:

```
kd> lm kv m myfault*
start    end        module name
f224d000 f224dbe0   myfault      (deferred)
    Image path: \??\C:\WINNT\system32\drivers\myfault.sys
    Timestamp: Thu Apr 29 14:53:12 2004 (40915D28)  Checksum: 00010090
    ImageSize : 00000BE0
    File version:     2.0.0.0
    Product version:  2.0.0.0
    File flags:       0 (Mask 3F)
    File OS:          40004 NT Win32
    File type:        3.7 Driver
    File date:        00000000.00000000
    Translations:     0409.04b0
    CompanyName:      Sysinternals
    ProductName:      Sysinternals Myfault
    InternalName:     myfault.sys
    OriginalFilename: myfault.sys
    ProductVersion:   2.0
    FileVersion:      2.0
    FileDescription:  Crash Test Driver
    LegalCopyright:   Copyright (C) M. Russinovich 2002-2004
```

Вы можете идентифицировать назначение драйвера по описанию, а также выяснить по версии файла и продукта, установлена ли у вас самая последняя версия. (Это можно определить, например, посетив сайт разработчика драйвера.) Если информация о версии отсутствует, например в момент краха соответствующая страница была выгружена из физической памяти, вы получите ее из свойств файла образа драйвера: просмотрите их с помощью Windows Explorer.

Средства анализа проблем, вызывающих крах

В предыдущем разделе, когда мы вызвали крах системы, выбрав параметр High IRQL Fault (Kernelmode) в Notmyfault, автоматический анализ дампа в отладчике не составил труда. Увы, в большинстве случаев исследовать крах системы с помощью отладчика сложно, а зачастую и невозможно. Существует несколько уровней верификации (с нарастающей степенью сложности и пропорциональным падением производительности системы), которые позволяют добиться того, чтобы вместо дампа, непригодного для анализа, генерировался дамп, пригодный для анализа. Если после настройки системы в соответствии с требованиями одного уровня и перезагрузки, вам не удалось выявить причину краха, попробуйте перейти на следующий уровень.

1. Если вы считаете, что крах системы может вызывать один или несколько драйверов, поскольку они были установлены в систему относительно недавно или их недавно обновили, или это следует из обстоятельств, при которых система терпит крах, то включите верификацию этих драйверов в Driver Verifier и выберите все режимы верификации, кроме имитации нехватки ресурсов.
2. Задайте тот же уровень верификации, но для всех неподписанных драйверов в системе. Или, если вы работаете с Windows 2000, в которой Driver Verifier не делает различий между подписанными и неподписанными драйверами, включите верификацию всех драйверов, поставляемых не Microsoft, а другими компаниями.
3. Задайте тот же уровень верификации, но для всех драйверов системы. Чтобы сохранить приемлемую производительность, можно разбить драйверы на группы и в промежутках между перезагрузками активизировать Driver Verifier для какой-то одной группы драйверов.

Очевидно, прежде чем тратить время и силы на изменение конфигурации системы и анализ аварийных дампов, стоит убедиться в том, что используются последние версии компонентов ядра и драйверов сторонних поставщиков, и при необходимости обновить их через Windows Update или напрямую через сайты производителей устройств.

> **ПРИМЕЧАНИЕ** Если загрузка вашей системы стала невозможной из-за того, что Driver Verifier обнаруживает ошибку драйвера и вызывает крах системы, загрузите систему в безопасном режиме (в котором верификация отключена), запустите Driver Verifier и отключите параметры проверки.

В следующих разделах показывается, как с помощью Driver Verifier сделать так, чтобы вместо дампов, непригодных для отладки, создавались дампы, позволяющие решить проблему. Кроме того, почитайте справочный файл Debugging Tools, где есть руководства по методикам углубленной отладки.

Переполнение буфера и особый пул

Несомненно, что чаще всего причиной краха Windows является повреждение пула. Обычно оно вызывается ошибкой драйвера, в результате которой данные записываются до начала или за концом буфера, выделенного в пуле подкачиваемой или неподкачиваемой памяти. Структуры управления пулами (pool tracking structures) исполнительной системы располагаются с каждой стороны буфера и отделяют их друг от друга. Таким образом, подобные ошибки приводят к повреждению структур управления пулами, повреждению буферов других драйверов или и к тому, и к другому. Крах, вызванный повреждением пулов, практически невозможно исследовать с помощью отладчика, поскольку крах системы происходит при обращении к поврежденным данным, а не в момент их повреждения.

> **ПРИМЕЧАНИЕ** Чтобы облегчить выявление этих трудноуловимых повреждений, в Windows XP Service Pack 2 (или выше) всегда выполняется проверка выхода за границы блока в пуле (pool-block tail checking). Поэтому переполнение буфера скорее всего тут же приведет к краху BAD_POOL_HEADER.

Вы можете вызвать крах, связанный с переполнением буфера, запустив Notmyfault и выбрав переключатель Buffer Overflow. В этом случае Myfault выделит память под буфер и перезапишет 40 байтов, идущих после буфера. Между щелчком кнопки Do Bug и крахом системы может пройти довольно много времени, возможно, вам даже придется задействовать пул, запустив какие-либо приложения. Это еще раз подчеркивает, что повреждение может не скоро привести к последствиям, влияющим на стабильность системы. Анализ аварийного дампа, полученного при такой ошибке, почти всегда показывает, что проблема связана с Ntoskrnl или каким-либо другим драйвером. И это демонстрирует бесполезность детального анализа при таком описании стоп-кода:

```
DRIVER_CORRUPTED_EXPOOL (c5)
An attempt was made to access a pageable (or completely invalid) address at an
interrupt request level (IRQL) that is too high.  This is caused by drivers that
have corrupted the system pool.  Run the driver verifier against any new (or
suspect) drivers, and if that doesn't turn up the culprit, then use gflags to
enable special pool.
Arguments:
Arg1: 4f4f4f53, memory referenced
Arg2: 00000002, IRQL
Arg3: 00000001, value 0 = read operation, 1 = write operation
Arg4: 80467139, address which referenced memory
```

В описании стоп-кода рекомендуется запустить Driver Verifier для каждого нового или подозрительного драйвера или активизировать особый пул с помощью Gflags. В обоих случаях преследуется одна и та же цель: выявить потенциальное повреждение в момент, когда оно происходит, и вызвать

крах системы так, чтобы при автоматическом анализе удалось обнаружить драйвер, вызвавший повреждение.

Если в Driver Verifier включен режим особого пула, проверяемые драйверы используют специальный пул вместо пула подкачиваемой или неподкачиваемой памяти во всех случаях, когда выделяется память для буферов размера, немного меньшего размера страницы. Буфер, память под который выделяется из особого пула, заключен между двумя недействительными страницами и по умолчанию выравнивается по верхней границе страницы. Кроме того, подпрограммы управления особым пулом заполняют неиспользуемое пространство страницы, содержащей буфер, по случайному шаблону. На рис. 14-8 показано, как выделяется память из особого пула.

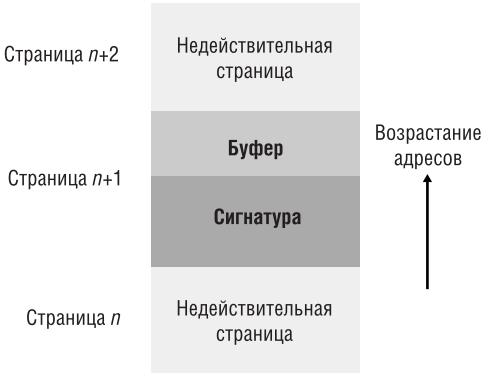


Рис. 14-8. *Выделение памяти из особого пула*

Система обнаруживает любые переполнения буфера, содержащегося в странице, поскольку они приводят к ошибке страницы: происходит обращение к недействительной странице, которая идет за буфером. Сигнатура нужна, чтобы перехватывать выход за конец буфера в момент, когда драйвер освобождает буфер: при выходе за конец будет нарушена целостность шаблона, помещенного в эту область при выделении памяти под буфер.

Чтобы посмотреть, как с помощью особого пула вызвать крах системы, который легко диагностировать с помощью механизма автоматического анализа, запустите Driver Verifier Manager (Диспетчер проверки драйверов). В Windows 2000 перейдите на вкладку Settings (Параметры), введите **myfault.sys** в текстовое поле внизу страницы, предназначенное для задания дополнительных драйверов, установите флажок особого пула, сохраните изменения, выйдите из Driver Verifier Manager и перезагрузитесь. В Windows XP и Windows Server 2003 выберите Create Custom Settings (For Code Developers) [Создать не стандартные параметры (для кода программ)] на первой странице мастера, на второй — Select Individual Settings From A Full List (Вы-брать параметры из списка), на третьей — Special Pool (Особый пул). Далее выберите Select Drivers From A List (Выберите имя драйвера из списка), а на странице, где перечислены типы драйверов, введите **myfault.sys** в диалоговом окне, открываемом после нажатия кнопки добавления незагруженных драйверов. (Не ищите в

этом диалоговом окне файл myfault.sys — просто введите его имя.) Затем отметьте драйвер myfault.sys, выйдите из мастера и перезагрузитесь.

Когда вы запустите Notmyfault и вызовете переполнение буфера, сразу же произойдет крах системы, а анализ дампа даст следующий результат:

```
Probably caused by : myfault.sys ( myfault+3f1 )
```

При детальном анализе вы получите следующее описание стоп-кода:

```
DRIVER_PAGE_FAULT_BEYOND_END_OF_ALLOCATION (d6)
N bytes of memory was allocated and more than N bytes are being referenced. This
cannot be protected by try-except. When possible, the guilty driver's name
(Unicode string) is printed on the bugcheck screen and saved in KiBugCheckDriver.
Arguments:
Arg1: beb50000, memory referenced
Arg2: 00000001, value 0 = read operation, 1 = write operation
Arg3: ec3473f1, if non-zero, the address which referenced memory.
Arg4: 00000000, (reserved)
```

Благодаря особому пулу трудноуловимая ошибка немедленно проявила себя, и анализ стал тривиальным.

Перезапись кода и защита системного кода от записи

Драйвер, в котором из-за «бага» происходит повреждение или неправильная интерпретация его собственных структур данных, может обращаться к не принадлежащей ему памяти, воспринимая поврежденные данные как указатель на область памяти. Такой некорректный указатель может указывать на что угодно в адресном пространстве, в том числе на данные, принадлежащие другим драйверам, недействительные страницы памяти или на код других драйверов или ядра. Как и при переполнении буфера, драйвер, вызвавший повреждение данных, обычно не удается идентифицировать в момент, когда повреждение обнаруживается и происходит крах системы. Использование особого пула увеличивает вероятность выявления «багов», связанных с некорректными указателями, но не выявляет повреждение кода.

Если вы запустите Notmyfault и выберете переключатель Code Overwrite, драйвер Myfault повредит точку входа функции *NtReadFile*. Далее возможны два варианта. Если ваша система работает под управлением Windows 2000 и оснащена не более чем 127 Мб физической памяти или работает под управлением Windows XP или Windows Server 2003 и оснащена не более чем 255 Мб физической памяти, произойдет крах и анализ дампа укажет на Myfault.sys.

В описании стоп-кода, выводимом при детальном анализе, говорится, что драйвер Myfault попытался записать данные в память, доступную только для чтения:

```
ATTEMPTED_WRITE_TO_READONLY_MEMORY (be)
An attempt was made to write to readonly memory.  The guilty driver is on the
stack trace (and is typically the current instruction pointer). When possible, the
guilty driver's name (Unicode string) is printed on the bugcheck screen and saved
in KiBugCheckDriver.
```

```
Arguments:
Arg1: 804bb7fd, Virtual address for the attempted write.
Arg2: 004bb121, PTE contents.
Arg3: b804db60, (reserved)
Arg4: 0000000b, (reserved)
```

Однако, если у вас Windows 2000 и более 127 Мб памяти либо Windows XP или Windows Server 2003 и более 255 Мб памяти, произойдет крах другого типа, так как повреждение памяти сразу не проявится. Поскольку *NtReadFile* — широко используемая системная функция, к которой подсистема Windows обращается при считывании ввода с клавиатуры или от мыши, крах системы произойдет почти сразу же, как только какой-либо поток попытается выполнить поврежденный код. Возникнет ошибка из-за выполнения недопустимой инструкции. Анализ аварийного дампа, выполняемый в этом случае, может давать разные результаты, но они обязательно будут неправильными. Обычно механизм анализа приходит к выводу, что наиболее вероятными источниками ошибки являются Windows.sys или Ntoskrnl.exe. При таком крахе выводится следующее описание стоп-кода:

```
KMODE_EXCEPTION_NOT_HANDLED (1e)
This is a very common bugcheck. Usually the exception address pinpoints the
driver/function that caused the problem.  Always note this address as well as the
link date of the driver/image that contains this address.
Arguments:
Arg1: c0000005, The exception code that was not handled
Arg2: 80461885, The address that the exception occurred at
Arg3: 00000000, Parameter 0 of the exception
Arg4: 00000000, Parameter 1 of the exception
```

Разные конфигурации ведут себя по-разному в связи с тем, что в Windows 2000 введен механизм *защиты системного кода от записи* (system code write protection). В таблице 14-2 показано, в каких конфигурациях защита системного кода от записи не используется по умолчанию.

Таблица 14-2. *Конфигурации, в которых отключена защита системного кода от записи*

	Windows 2000	Windows XP и Windows Server 2003
Защита системного кода от записи отключена	RAM > 127 Мб	RAM > 255 Мб

Если защита системного кода от записи включена, диспетчер памяти проецирует Ntoskrnl.exe, HAL и загрузочные драйверы как стандартные физические страницы (4 Кб для x86 и x64, 8 Кб для IA64). Поскольку при проецировании образов обеспечивается детализация с точностью до размера стандартной страницы, диспетчер памяти может защитить страницы, содержащие код, от записи и генерировать ошибку доступа при попытке их модификации (что вы и видели при первом крахе). Но когда защита системного кода от записи отключена, диспетчер памяти использует при проецировании Ntoskrnl.exe большие страницы (4 Мб для x86 или 16 Мб для IA64 и x64). Та-

кой режим по умолчанию действует в Windows 2000 при наличии более чем 127 Мб памяти, а в Windows XP или Windows Server 2003 — при наличии более чем 255 Мб памяти. Диспетчер памяти не может защитить код, поскольку код и данные могут находиться на одной странице.

Если защита системного кода от записи отключена и при анализе аварийного дампа сообщается о маловероятных причинах краха или если вы подозреваете, что произошло повреждение кода, следует включить защиту. Для этого проще всего включить проверку хотя бы одного драйвера с помощью Driver Verifier. Кроме того, можно включить защиту вручную, добавив два параметра в раздел реестра HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management. Сначала укажите максимально возможное значение для объема памяти, начиная с которого диспетчер памяти использует при проецировании Ntoskrnl.exe большие страницы вместо стандартных. Создайте параметр LargePageMinimum типа DWORD, присвойте ему значение 0xFFFFFFFF. Добавьте еще один параметр типа DWORD — EnforceWriteProtection — и присвойте ему значение 1. Чтобы изменения вступили в силу, перезагрузите компьютер.

ПРИМЕЧАНИЕ Когда отладчик имеет доступ к файлам образов, включенным в аварийный дамп, при анализе на внутреннем уровне выполняется команда отладчика *!chkimg*, которая проверяет, совпадает ли копия образа в аварийном дампе с образом на диске, и сообщает о различиях. Заметьте: если вы активизируете Driver Verifier, *chkimg* обязательно обнаружит различия при сравнении с файлом Ntoskrnl.exe.

Углубленный анализ аварийных дампов

В предыдущем разделе рассказывалось о том, как с помощью Driver Verifier получать аварийные дампы, автоматический анализ которых может решить проблему. Тем не менее, возможны случаи, когда невозможно добиться, чтобы система сгенерировала дамп, который легко проанализировать. В таких случаях нужен анализ вручную, чтобы попытаться определить, в чем заключается проблема.

- С помощью команды отладчика *!process 0 0* посмотрите, какие процессы выполняются, и убедитесь, что вам понятно назначение каждого из них. Попробуйте завершить или удалить приложения и сервисы, без которых можно обойтись.
- С помощью команды *lm* с параметром *kv* выведите список загруженных драйверов режима ядра. Убедитесь, что вам понятно назначение каждого из драйверов сторонних поставщиков и что вы используете самые последние версии.
- С помощью команды *!vm* проверьте, не исчерпаны ли виртуальная память системы, пул подкачиваемой памяти и пул неподкачиваемой памяти. Если исчерпана виртуальная память, объем переданных страниц будет близок

к пределу. В этом случае попытайтесь выявить потенциальную утечку памяти: просмотрите список процессов и выберите те из них, которым передано много памяти. Если исчерпан пул подкачиваемой или неподкачиваемой памяти (т. е. объем занятой памяти близок к максимуму), см. эксперимент «Анализ утечки памяти в пуле» в главе 7.

Существуют и другие отладочные команды, которые могут оказаться полезными, но для их применения нужны более глубокие знания. Одной из таких команд является *!irp*. В следующем разделе показано, как с ее помощью идентифицировать подозрительные драйверы.

Засорение стека

Переполнение или засорение стека (stack trashing) вызывается ошибками, связанными с выходом за конец или начало буфера. Однако в таких случаях буфер находится не в пуле, а в стеке потока, выполняющего ошибочный код. Ошибки этого типа также трудны в отладке, поскольку стек играет важную роль при любом анализе аварийного дампа.

Когда вы запускаете Notmyfault и выбираете Stack Trash, драйвер Myfault переполняет буфер, память под который выделена в стеке потока, где выполняется код драйвера. Myfault пытается вернуть управление вызвавшей его функции Ntoskrnl и считывает из стека адрес возврата, с которого должно продолжиться выполнение. Однако этот адрес поврежден при переполнении буфера стека, поэтому поток продолжает выполнение с какого-то другого адреса, может быть, даже не содержащего код. Когда поток попытается выполнить недопустимую инструкцию процессора или обратится к недопустимой области памяти, будет сгенерировано исключение и произойдет крах системы.

В различных случаях краха анализ аварийного дампа, проводимый при переполнении стека, будет указывать на разные драйверы, но стоп-код всегда будет одним и тем же — KMODE_EXCEPTION_NOT_HANDLED. Если вы выполните детальный (verbose) анализ, трассировочная информация для стека будет выглядеть так:

```
STACK_TEXT:
b7b0ebd4 00000000 00000000 00000000 00000000 0x0
```

Это объясняется тем, что мы перезаписываем стек нулями. К сожалению, такие механизмы, как особый пул и защита системного кода от записи, не позволяют выявлять «баги» этого типа. Придется выполнять анализ вручную, по косвенным признакам определяя, какой драйвер выполнялся в момент повреждения стека. Один из возможных вариантов — исследовать IRP-пакеты, с которыми работает поток, выполняемый в момент засорения стека. Когда поток передает запрос ввода-вывода, диспетчер ввода-вывода записывает указатель на соответствующий IRP в список Irp, хранящийся в структуре ETHREAD потока. Команда отладчика *!thread* выводит дамп этого списка для заданного потока. (Если адрес объекта «поток» не указан, команда *!thread*

выводит дамп для текущего потока, выполняемого процессором.) Затем IRP можно изучить с помощью команды *!irp*:

```
kd> !thread
THREAD ff740020  Cid 8f8.420  Teb: 7ffde000  Win32Thread: a20cdbe8
 RUNNING
IRP List:
    bc5a7f68: (0006,0094) Flags: 00000000  Mdl: 00000000
Not impersonating
Owning Process ff75f120
...

kd> !irp bc5a7f68
Irp is active with 1 stacks 1 is current (= 0xbc5a7fd8)
 No Mdl Thread ff740020:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
>[  e, 0]   0  0 ff79e4c0 ff7ac028 00000000-00000000
       \Driver\MYFAULT Args: 00000000 00000000 83360010 00000000
```

Вывод показывает, что текущий и единственный фрагмент стека для IRP (обозначенный префиксом «>») принадлежит драйверу Myfault. Если бы это было на практике, далее следовало бы убедиться, что установлена последняя версия драйвера, и, если это не так, установить новую версию. Если это не помогло, нужно было бы активизировать Driver Verifier для данного драйвера (включив все режимы, кроме имитации нехватки памяти).

Зависание или отсутствие отклика системы

Если система перестает отвечать (т. е. не реагирует на ввод с клавиатуры или мыши, курсор мыши не перемещается или вы можете перемещать курсор, но система не реагирует на щелчки), говорят, что система *зависла*. Существует несколько возможных причин зависания системы:

- при обращении к драйверу устройства ISR (interrupt service routine) или DPC не вернула управление;
- поток с высоким приоритетом (выполняемый в режиме реального времени) вытеснил потоки ввода данных в подсистему управления окнами (windowing system);
- произошла взаимная блокировка при выполнении кода в режиме ядра (два потока или процессора удерживают ресурсы, нужные друг другу, причем ни один из них не освобождает свой ресурс).

Если вы работаете с Windows XP или Windows Server 2003, то можете выявлять взаимные блокировки, используя одну из функций Driver Verifier — обнаружение взаимных блокировок (deadlock detection). При обнаружении взаимных блокировок ведется наблюдение за спин-блокировками (spin locks), быстрыми и обычными мьютексами и выявляются закономерности, которые могут приводить к взаимной блокировке. (Информацию об этих и других синхронизирующих примитивах см. в главе 3.) Если обнаружена такая ситу-

ация, Driver Verifier вызывает крах системы, указывая, какой драйвер является причиной взаимной блокировки. Простейшая форма взаимной блокировки — каждый из двух потоков удерживает некий ресурс, нужный другому потоку, при этом ни один из них не освобождает свой ресурс и ждет освобождения другого ресурса. Если вы используете Windows XP или Windows Server 2003, первое, что нужно сделать для устранения зависаний системы, — включить обнаружение взаимных блокировок для подозрительных драйверов, затем для неподписанных драйверов, а затем для всех драйверов. В этом режиме следует работать до тех пор, пока не произойдет крах системы, который позволит выявить драйвер, вызывающий взаимную блокировку.

Если вы используете Windows 2000 или если вы проверили все драйверы, а система продолжает зависать, то должны либо вручную вызвать крах зависшей системы и проанализировать полученный в результате дамп, либо исследовать систему с помощью отладчика ядра.

Итак, есть два подхода к исследованию зависающей системы, позволяющие выявить драйвер или компонент, который вызывает зависания. Первый — вызвать крах зависшей системы и надеяться, что будет получен дамп, который удастся проанализировать. Второй — исследовать систему с помощью отладчика ядра и проанализировать работу системы. И при том, и при другом подходе необходимы предварительная настройка и перезагрузка. Чтобы выявить и устранить причину зависания, в обоих случаях выполняется одно и то же исследование состояния системы.

Чтобы вручную вызвать крах зависшей системы, сначала добавьте в реестр параметр HKLM\System\CurrentControlSet\Services\i8042prt\Parameters\CrashOnCtrlScroll типа DWORD со значением 1. После перезагрузки порт-драйвер i8042, который является драйвером порта ввода с PS/2-клавиатуры, будет наблюдать за нажатиями клавиш в своей ISR (об ISR подробно рассказывается в главе 3) и отслеживать двукратное нажатие клавиши Scroll Lock при нажатой правой клавише Ctrl. Обнаружив такую последовательность нажатий, драйвер вызывает функцию *KeBugCheckEx* со стоп-кодом MANUALLY_INITIATED_CRASH (0xE2), указывающим, что крах инициирован пользователем вручную. Когда система перезагрузится, откройте аварийный дамп и с помощью методик, описанных выше, попробуйте установить, почему система зависла (например, определите, какой поток выполнялся, когда система зависла, попытайтесь понять, что произошло, проанализировав стек ядра и т. д.). Заметьте: этот подход работает в большинстве случаев зависания систем, но не годится, когда ISR порт-драйвера i8042 не выполняется. (Эта ISR не выполняется, если все процессоры зависли из-за того, что их IRQL выше, чем IRQL у ISR, или если повреждение системных структур данных затронуло код либо данные, используемые при обработке прерываний.)

ПРИМЕЧАНИЕ Вызов краха зависшей системы вручную на основе функциональности порт-драйвера i8042 невозможен при использовании USB-клавиатур. Этот подход работает только в случае PS/2-клавиатур.

Еще один способ вручную вызвать крах системы — использовать встроенную кнопку «crash». (Она имеется на некоторых серверах класса «high end».) Тогда, чтобы инициировать крах, материнская плата системы генерирует NMI (немаскируемое прерывание). Чтобы активизировать эту функцию, задайте значение 1 для содержащегося в реестре DWORD-параметра HKLM\System\CurrentControlSet\Control\CrashControl\NMICrashDump. В этом случае при нажатии кнопки «crash» в системе будет генерироваться NMI, и обработчик NMI-прерываний ядра вызовет *KeBugCheckEx*. Такой подход более универсален, чем применение порт-драйвера i8042, поскольку IRQL у NMI всегда выше, чем у прерывания порт-драйвера i8042. Дополнительные сведения см. по ссылке *http://www.microsoft.com/hwdev/platform/proc/dmpsw.asp*.

Если сгенерировать аварийный дамп вручную нельзя, попытайтесь исследовать зависшую систему. Прежде всего загрузите систему в отладочном режиме. Это можно сделать двумя способами. Нажмите клавишу F8 во время загрузки и выберите Debugging Mode (Режим отладки) или добавьте запись, задающую загрузку в отладочном режиме, в файл Boot.ini: скопируйте запись, которая уже имеется в файле Boot.ini системы, и добавьте ключ /DEBUG. При нажатии F8 система будет использовать соединение по умолчанию (последовательный порт COM2 и скорость 19200 бод). При использовании режима /DEBUG вы должны будете настроить механизм соединения между хост-системой, на которой выполняется отладчик ядра, и целевой системой, загружаемой в отладочном режиме, и задать ключи /Debugport и /Baudrate, соответствующие типу соединения. Доступно два типа соединения: нуль-модемный кабель, соединяющий последовательные порты, или (в системах Windows XP и Windows Server 2003) кабель IEEE 1394 (Firewire), подключенный к порту 1394 каждой системы. Подробности настройки хост-системы и целевой системы для отладки ядра см. в справочном файле Windows Debugging Tools.

При загрузке в отладочном режиме система загружает отладчик ядра и готовит его к соединению с отладчиком ядра, выполняемом на другом компьютере, подключенном по нуль-модемному кабелю или по IEEE 1394. Заметьте: присутствие отладчика ядра не влияет на производительность. Когда система зависнет, запустите отладчик Windbg или Kd на подключенной системе, установите соединение между отладчиками ядра и выполните отладку кода зависшей системы. Такой подход не сработает, если прерывания отключены или если поврежден код отладчика ядра.

ПРИМЕЧАНИЕ Загрузка системы в отладочном режиме не влияет на производительность, если эта система не соединена с другой. Однако этого нельзя сказать о системе, настроенной на автоматическую перезагрузку после краха: если при загрузке системы включена отладка ядра, то после краха системы отладчик ядра будет ожидать соединения с другой системой.

При выполнении анализа можно не оставлять систему в остановленном состоянии, а с помощью команды отладчика *.dump* создать файл аварийного

дампа на хост-компьютере отладки. Затем перезагрузить зависшую систему и проанализировать аварийный дамп в автономном режиме (или отправить его в Microsoft). Заметьте: это может занять много времени, если вы используете нуль-модемный кабель (по сравнению с более скоростным соединением 1394), поэтому можно получить только минидамп командой *.dump /m*. Если целевой компьютер способен записать аварийный дамп, можно заставить его сделать это, введя в отладчике команду *.crash*. Тогда целевой компьютер создаст дамп на своем локальном жестком диске, и вы сможете посмотреть дамп после перезагрузки системы.

Зависание можно вызвать, запустив Notmyfault и выбрав параметр Hang. Тогда драйвер Myfault поставит в очередь DPC, выполняющую бесконечный цикл для каждого процессора системы. Поскольку при выполнении DPC-функций IRQL процессора имеет уровень «DPC/dispatch», ISR клавиатуры будет реагировать на последовательность нажатий клавиш, вызывающую крах.

Когда вы приступили к отладке зависшей системы или загрузили в отладчик дамп, который вручную сгенерировали для зависшей системы, следует выполнить команду *!analyze* с параметром *–hang*. Тогда отладчик проанализирует блокировки системы и попытается определить, не произошла ли взаимная блокировка, и, если да, то какой драйвер или драйверы в ней участвуют. Однако, если зависание аналогично вызванному программой Notmyfault, команда *!analyze* не сообщит ничего полезного.

Если команда *!analyze* не помогла решить проблему, выполните команды *!thread* и *!process* в каждом из контекстов процессоров для дампа. (Для переключения между контекстами процессоров используйте команду ~, например *~1* переключает в контекст процессора 1.) Если поток, вызвавший зависание системы, выполняет бесконечный цикл на уровне IRQL «DPC/dispatch» или выше, вы увидите модуль драйвера, в котором это происходит, в трассировочной информации стека, выводимой командой *!thread*. Если зависание системы вызвано программой Notmyfault, трассировочная информация стека, получаемая по аварийному дампу системы, выглядит так:

```
STACK_TEXT:
f9e66ed8 f9b0d681 000000e2 00000000 00000000 nt!KeBugCheckEx+0x19
f9e66ef4 f9b0cefb 0069b0d8 010000c6 00000000
 i8042prt!I8xProcessCrashDump+0x235
f9e66f3c 804ebb04 81797d98 8169b020 00010009
 i8042prt!I8042KeyboardInterruptService+0x21c
f9e66f3c fa12e34a 81797d98 8169b020 00010009
 nt!KiInterruptDispatch+0x3d
WARNING: Stack unwind information not available.
Following frames may be wrong.
ffdff980 8169b288 f9e67000 0000210f 00000004 myfault+0x34a
8054ace4 ffdff980 804ebf58 00000000 0000319c 0x8169b288
8054ace4 ffdff980 804ebf58 00000000 0000319c 0xffdff980
8169ae9c 8054ace4 f9b12b0f 8169ac88 00000000 0xffdff980
...
```

Первые несколько строк трассировочной информации стека относятся к подпрограммам, вызванным, когда вы нажали клавиши, по которым порт-драйвер i8042 вызывает крах системы. Присутствие драйвера Myfault означает, что зависание системы могло произойти из-за него.

Еще одна команда, которая может оказаться полезной, — *!locks*; она выводит состояние всех блокировок ресурсов исполнительной системы. По умолчанию команда показывает только *спорные* ресурсы, т. е. ресурсы, на владение которыми претендуют минимум два потока. Исследуйте стеки потоков, владеющих такими ресурсами, с помощью команды *!thread*, и посмотрите, какому драйверу они могут принадлежать.

Если аварийного дампа нет

В этом разделе мы рассмотрим, как устранять неполадки в системах, которые по каким-либо причинам не записывают аварийный дамп. Аварийный дамп может не записываться из-за того, что размер страничного файла на загрузочном томе слишком мал, чтобы вместить дамп, или из-за того, что на диске недостаточно места, чтобы извлечь дамп после перезагрузки. Эти две причины легко устранить, увеличив размер страничного файла или задав при настройке, что дамп сохраняется на томе, где достаточно места.

Третьей причиной, по которой аварийный дамп не записывается, может быть то, что код ядра и структуры данных, необходимые для записи аварийного дампа, повреждены при крахе. Как уже говорилось, для этих данных подсчитывается контрольная сумма, и, если при крахе обнаружено несовпадение контрольных сумм, система даже не пытается сохранить аварийный дамп (чтобы не рисковать данными на диске). Поэтому в таком случае нужно отслеживать момент краха системы и пытаться определить причину краха.

Наконец, еще одна причина в том, что дисковая подсистема не может обрабатывать запросы записи на диск (ситуация, которая сама по себе может вызвать сбой системы). Такая ситуация возникает, если произошел аппаратный сбой контроллера дисков или поврежден кабель жесткого диска.

Одно из простых решений — отключить параметр Automatically Restart (Выполнить автоматическую перезагрузку) в параметрах Startup And Recovery (Загрузка и восстановление), чтобы можно было изучать «синий экран» с консоли. Однако текст «синего экрана» позволяет выявить причины краха системы только в самых простых случаях.

Для более глубокого анализа необходимо с помощью отладчика ядра исследовать поведение системы в момент краха. Для этого загрузите систему в отладочном режиме, о котором рассказывалось в предыдущем разделе. Когда происходит крах системы, загруженной в отладочном режиме, она не выводит «синий экран» и не пытается записать дамп, а ожидает соединения с отладчиком ядра, выполняемым на хост-системе. Поэтому можно увидеть, что вызвало причину краха, и, вполне вероятно, провести некий базовый анализ с помощью команд отладчика ядра, описанных ранее. Как говорилось в предыдущем разделе, команда отладчика позволяет сохранить копию па-

мяти системы, потерпевшей крах, для дальнейшей отладки, что даст возможность перезагрузить эту систему и вести отладку в автономном режиме.

ЭКСПЕРИМЕНТ: экранная заставка Blue Screen

Отличный способ вспомнить, как выглядит «синий экран», или подшутить над своими друзьями и коллегами — запустить экранную заставку Sysinternals Blue Screen, которую можно скачать с сайта *wwwsysinternals.com*. Она точно имитирует «синий экран» для той версии Windows, в которой вы работаете, и выводит системную информацию (например, список загруженных драйверов), соответствующую действительности. Кроме того, она имитирует автоматическую перезагрузку, показывая экран запуска Windows. Заметьте: в отличие от других экранных заставок, исчезающих при перемещении мыши, Blue Screen требует нажатия клавиши.

С помощью утилиты Psexec с сайта Sysinternals вы даже можете запустить экранную заставку на другой системе, выполнив команду:

```
psexec \\computername -i -d "c:\sysinternals bluescreen.scr" -s
```

Для этого у вас должны быть административные привилегии на удаленной системе. (С помощью ключей *–u* и *–p* утилиты Psexec можно задать другие удостоверения защиты.) Проверьте, есть ли у ваших коллег чувство юмора!

Словарь терминов

A

access-control list (ACL) — список управления доступом (ACL) ~ часть дескриптора защиты, в которой перечисляется, кто имеет доступ к объекту и какой. Владелец объекта может изменять ACL этого объекта, разрешая или запрещая доступ к нему другим пользователям. ACL состоит из заголовка и произвольного числа элементов ACL (ACE). ACL без ACE называется пустым (null ACL) и указывает, что доступ к объекту не разрешен ни одному пользователю.

access token — маркер доступа ~ структура данных, определяющая характеристики защиты процесса или потока и включающая идентификатор защиты (security ID, SID), список групп, в которые входит пользователь, и список привилегий. У каждого процесса имеется основной маркер доступа (primary access token), по умолчанию наследуемый от родительского процесса.

add-device routine — процедура добавления устройства ~ процедура, реализуемая драйверами, которые поддерживают Plug and Play. Через эту процедуру диспетчер Plug and Play уведомляет драйвер при обнаружении соответствующего устройства. В этой процедуре драйвер, как правило, создает объект «устройство», представляющий обнаруженное устройство.

Address Windowing Extensions (AWE) ~ механизм Windows, позволяющий 32-разрядному приложению выделять до 128 Гб физической памяти и проецировать ее представления, или окна, на свое 2-гигабайтное виртуальное адресное пространство. Применение AWE усложняет программирование, но решает проблему прямого доступа к дополнительной физической памяти.

affinity mask — маска привязки (к процессорам) ~ битовая маска, определяющая, на каких процессорах разрешено выполнение данного потока. Начальная маска привязки потока к процессорам наследуется от его процесса.

alertable wait state — состояние «тревожного» ожидания ~ состояние потока, при котором он либо переходит к ожиданию на объекте и указывает, что это ожидание должно быть «тревожным» (вызовом Windows-функции *WaitForMultipleObjectsEx*), либо напрямую проверяет появление APC в своей очереди (используя *SleepEx*). В любом случае, если в очереди потока появляется APC пользовательского режима, ядро прерывает поток, передает управление APC-процедуре, а по завершении APC-процедуры возобновляет его выполнение. APC пользовательского режима доставляются потоку, только когда он находится в состоянии «тревожного» ожидания.

allocation granularity — гранулярность выделения памяти ~ дискретность выделения виртуальной памяти. Windows выравнивает каждый регион зарезервированного адресного пространства процесса так, чтобы он начинался с границы, кратной гранулярности выделения памяти. Последнее значение можно получить через Windows-функцию *GetSystemInfo*. В настоящее время оно составляет 64 Кб. Такое значение было выбрано с учетом возможности поддержки будущими процессорами страниц большого размера (например, до 64 Кб). (Код Windows режима ядра может резервировать память с гранулярностью в одну страницу памяти.)

APC queue — APC-очередь ~ очередь, в которую помещаются APC. Такие очереди (одна — для пользовательского режима, другая — для режима ядра) индивидуальны для каждого потока. (Заметьте, что DPC-очередь индивидуальна для каждого процессора.)

asymmetric multiprocessing (ASMP) — асимметричная мультипроцессорная обработка ~ схема обработки, при которой один процессор обычно выделяется коду операционной системы, а остальные процессоры выполняют только пользовательский код.

asynchronous I/O — асинхронный ввод-вывод ~ модель ввода-вывода, которая позволяет приложению выдать запрос на ввод-вывод и перейти к выполнению других задач на то время, пока запрос обрабатывается.

asynchronous procedure call (APC) ~ функция, которая позволяет пользовательским программам и системе выполнять код в контексте конкретного пользовательского потока (т. е. в адресном пространстве конкретного процесса). APC могут быть либо пользовательского режима, либо режима ядра. Последние (в отличие от первых) не требуют «разрешения» целевого потока на выполнение в его контексте.

asynchronous read-ahead with history — асинхронное опережающее чтение с хронологией ~ метод, используемый диспетчером кэша для сохранения хронологии последних двух запросов на чтение в закрытой карте кэша применительно к описателю файла, к которому происходит обращение.

atomic transaction — атомарная транзакция ~ метод внесения изменений в базу данных, не влияющий на корректность информации или целостность базы данных в случае сбоя системы. В основе атомарных транзакций лежит принцип «все или ничего»: если хотя бы одна из операций с базой данных заканчивается неудачно, отменяются все операции текущей атомарной транзакции. Если транзакция прерывается из-за сбоя системы, изменения, уже внесенные в ходе этой транзакции, подлежат отмене, или откату. Операция отката возвращает базу данных в предыдущее согласованное состояние. *См. также* transaction.

attribute list — список атрибутов ~ особый тип атрибута файла в заголовке NTFS-файла, который содержит дополнительные атрибуты. Список атрибутов создается в том случае, если у файла слишком много атрибутов и они не умещаются в записи главной таблицы файлов (MFT). Атрибут «список атрибутов» содержит имя и код типа всех атрибутов файла, а также файловые ссылки на записи MFT, в которых хранятся эти атрибуты.

authentication packages — пакеты аутентификации ~ DLL-модули, которые выполняются в контексте LSASS и реализуют политику аутентификации. DLL аутентификации отвечает за проверку имени и пароля пользователя и, если они верны, за передачу LSASS детальной информации об удостоверениях защиты данного пользователя. В Windows включено два пакета аутентификации: Kerberos и MSV1_0.

automatic working set trimming — автоматическое усечение рабочих наборов ~ метод, применяемый диспетчером памяти для увеличения объема свободной памяти в системе при нехватке физической памяти.

B

bad-cluster file — файл плохих кластеров ~ системный файл (с именем $BadClus), в котором регистрируются аварийные (плохие) участки дискового пространства на томе.

balance set manager — диспетчер настройки баланса ~ системный поток, который пробуждается раз в секунду для проверки и при необходимости для инициации различных событий, связанных с планированием потоков и управлением памяти.

basic disk — базовый диск ~ диск, разбитый на разделы в стиле MS-DOS. *См. также* dynamic disk.

bitmap file — файл битовой карты ~ системный файл (с именем $Bitmap), в котором NTFS регистрирует занятые и свободные кластеры на томе. Атрибут данных этого файла содержит битовую карту, каждый бит которой представляет один из кластеров тома и сообщает, свободен этот кластер или выделен какому-то файлу.

boot code — загрузочный (стартовый) код ~ команды, выполняемые при загрузке системы.

boot device drivers — загрузочные драйверы устройств (используемые на этапе загрузки системы) ~ драйверы устройств, необходимые для загрузки системы.

boot file — загрузочный файл ~ системный файл (с именем $Boot), в котором хранится код начальной загрузки Windows.

boot partition — загрузочный раздел ~ раздел, в котором хранятся основные файлы операционной системы. Загрузочный раздел идентифицируется системой при запуске. Код,

содержащийся в главной загрузочной записи (MBR), сканирует таблицу главных разделов и находит раздел с флагом, который сообщает, что данный раздел является загрузочным. Как только MBR находит хотя бы один такой флаг, она считывает первый сектор соответствующего раздела в память и передает управление коду из этого сектора.

boot sector — загрузочный сектор ~ первый сектор раздела, помеченного как активный. Из этого сектора загружается MBR (главная загрузочная запись). За-грузочный сектор содержит информацию, идентифицирующую формат и структуру файловой системы в данном разделе.

boot volume — загрузочный том ~ том, где содержатся операционная система Windows и ее вспомогательные файлы. Загрузочный том может совпадать с системным томом.

bus driver — драйвер шины ~ драйвер, обслуживающий контроллер шины, адаптер, мост и любое устройство, у которого имеются дочерние устройства. Такие драйверы обязательны, они обычно поставляются Microsoft; для каждой шины, например PCI, PCMCIA и USB, в системе должен быть один драйвер шины.

C

cache manager — диспетчер кэша ~ компонент исполнительной системы Windows, который предоставляет общесистемные сервисы кэширования для NTFS и других файловых систем, в том числе сетевых.

careful write — точная запись ~ метод поддержки файлового ввода-вывода и кэширования в системе. *См. также* write-through.

change journal — журнал изменений ~ внутренний файл, в котором NTFS регистрирует информацию, позволяющую приложениям вести эффективный мониторинг изменений в файлах и каталогах. Журнал изменений достаточно велик, чтобы приложения успевали обрабатывать все изменения.

checked build — проверочная версия ~ специальная отладочная версия Windows, доступная только при подписке на MSDN уровня Professional или Universal. Проверочная версия создается компиляцией исходного кода Windows с флагом DEBUG периода компиляции, установленным в TRUE.

checkpoint record — запись контрольной точки ~ запись, помогающая NTFS определять, какая обработка понадобится для восстановления тома, если система рухнет в данный момент. Эта запись также включает информацию для повтора и отмены.

class driver — драйвер класса ~ тип драйвера устройства режима ядра, реализующий обработку ввода-вывода для конкретного класса устройств, например для дисков, ленточных накопителей или приводов CD-ROM.

clock interrupt handler — обработчик прерываний от системного таймера ~ системная процедура, обновляющая системное время, а затем уменьшающая на 1 значение счетчика, который используется для отслеживания того, сколько времени выполняется текущий поток.

cluster factor — кластерный множитель ~ размер кластеров тома, устанавливаемый при форматировании конкретного тома командой *format* или с помощью оснастки Disk Management (Управление дисками) в Microsoft Management Console (MMC) (Консоль управления Microsoft).

cluster remapping — переназначение кластеров ~ процесс, в ходе которого NTFS динамически извлекает сохранившиеся данные из кластера с плохим сектором, выделяет новый кластер и копирует в него эти данные.

cluster — кластер ~ единица выделения дискового пространства на томе. Каждый кластер нумеруется 16-битным числом.

collided page fault — конфликт ошибок страницы ~ ситуация, когда другой поток или процесс вызывает ошибку той страницы, которая в данный момент загружается в память с диска.

commitment — передача ~ выделение физической памяти зарезервированному региону виртуальной памяти.

common model — общая модель ~ набор классов в Common Information Model (CIM), которые представляют объекты, специфичные для областей управления в системе, но независимые от конкретной реализации. Эти классы считаются расширением базовой модели CIM. *См. также* core model.

complete memory dump — полный дамп памяти ~ дамп памяти, включающий содержимое всей физической памяти на момент краха системы. Этот тип дампа требует, чтобы страничный файл был по крайней мере того же размера, что и физическая память. Windows NT 4 поддерживала только этот тип файлов аварийных дампов.

completion port — порт завершения ~ механизм уведомления потоков о завершении ввода-вывода. Как только файл сопоставляется с портом завершения, по окончании любой операции асинхронного ввода-вывода над этим файлом в очередь порта завершения ставится пакет завершения. Поток узнает о выполнении каких-либо операций ввода-вывода, просто проверяя наличие пакета завершения в очереди порта. Благодаря такому механизму число потоков, активно обслуживающих клиентские запросы, контролируется самой системой.

configuration manager — диспетчер конфигурации ~ один из основных компонентов исполнительной системы, отвечающих за реализацию системного реестра и управление им.

container object — объект-контейнер ~ объект пространства имен, способный хранить другие объекты, в том числе другие контейнеры. Примеры контейнеров — каталоги в пространстве имен файловой системы и разделы в пространстве имен реестра.

context switch — переключение контекста ~ процедура сохранения переменного состояния машины, связанного с выполняемым потоком, загрузки переменного состояния другого потока и начала выполнения этого потока.

control objects — управляющие объекты ~ набор объектов ядра, определяющих семантику управления различной функциональностью операционной системы. В этот набор входят объекты «процесс», APC- и DPC-объекты, а также несколько объектов, используемых подсистемой ввода-вывода, например объект прерывания.

core model — базовая модель ~ набор классов в Common Information Model (CIM), являющийся частью стандарта WBEM. Эти классы образуют базовый язык CIM и представляют объекты, применимые ко всем областям управления. *См. также* common model.

crash dump — аварийный дамп ~ запись содержимого системной памяти на момент краха, которая может помочь выяснить, какие компоненты вызвали крах.

critical section — критическая секция ~ взаимоисключающий синхронизирующий примитив, используемый для синхронизации доступа потоков внутри процесса к какому-либо общему ресурсу.

D

deferred procedure call (DPC) ~ процедура, выполняющая основную часть работы по обработке прерывания от устройства после выполнения процедуры обслуживания прерывания (interrupt service routine, ISR). DPC-процедура работает при уровне IRQL ниже того, на котором выполняется ISR, чтобы свести к минимуму блокирование других прерываний. DPC-процедура инициирует завершение ввода-вывода и начинает новую операцию ввода-вывода, ждущую в очереди устройства.

deferred procedure call (DPC) object — DPC-объект ~ управляющий объект ядра, который описывает запрос и позволяет отложить обработку прерывания до момента повышения IRQL до уровня «DPC/dispatch». Этот объект невидим программам пользовательского режима — он доступен только драйверам устройств и другому системному коду. Самая важная часть информации в DPC-объекте — адрес системной функции, которая будет вызвана ядром для обработки прерывания DPC.

deferred ready — состояние «готов, отложен» ~ состояние, используемое для потоков, выбранных для выполнения на конкретном процессоре, но пока не за-планированных к выполнению. Это новое состояние, введенное в Windows Server 2003, позволяет ядру свести к минимуму время, в течение которого удерживается общесистемная блокировка базы данных планировщика.

device drivers — драйверы устройств ~ загружаемые модули режима ядра (обычно SYS-файлы), образующие интерфейс между подсистемой ввода-вывода и соответствующим оборудованием. Драйверы устройств в Windows не работают с аппаратными устройствами напрямую, а вызывают функции HAL (уровня абстрагирования от оборудования).

device ID — идентификатор устройства ~ идентификатор, сообщаемый диспетчеру Plug and Play. Такие идентификаторы специфичны для конкретной шины; в случае шины USB он состоит из идентификатора изготовителя (vendor ID, VID) и идентификатора продукта (product ID, PID), назначенного изготовителем данному устройству.

device instance ID (DIID) — идентификатор экземпляра устройства (DIID) ~ идентификатор, который состоит из идентификатора устройства и идентификатора экземпляра и используется диспетчером Plug and Play для поиска раздела данного устройства в ветви реестра HKLM\SYSTEM\CurrentControlSet\Enum.

device object — объект «устройство» ~ структура данных, которая представляет физическое, логическое или виртуальное устройство в системе и описывает его характеристики.

device tree — дерево устройств ~ внутренняя структура данных, формируемая диспетчером Plug and Play и отражающая взаимосвязи между устройствами в системе. *См. также* devnode.

devnode — узел устройств ~ узел дерева устройств. Содержит информацию об объектах «устройство», которые представляют однотипные устройства и хранят информацию, специфическую для диспетчера Plug and Play. *См. также* device tree.

dirty page threshold — пороговое число измененных страниц ~ число страниц системного кэша в памяти, по достижении которого пробуждается поток подсистемы отложенной записи, принадлежащий диспетчеру кэша и вы-гружающий страницы кэша на диск. Это значение вычисляется при инициализации системы и зависит от объема физической памяти и параметра LargeSystemCache в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management.

discretionary access control — управление избирательным доступом ~ позволяет владельцу ресурса определять, кто может обращаться к данному ресурсу и что с ним можно делать. Владелец выдает права, разрешающие те или иные виды доступа, пользователю либо группе.

disk group — дисковая группа ~ динамические диски с общей базой данных. Windows поддерживает дисковые группы, но диспетчер логических дисков (Logical Disk Manager, LDM), реализованный в Windows, позволяет создать только одну дисковую группу.

dispatch code — код диспетчеризации, диспетчерский код ~ ассемблерный код, записываемый в объект прерывания при его инициализации. Этот код выполняется, когда возникает какое-нибудь прерывание.

dispatch routines — процедуры диспетчеризации, диспетчерские процедуры ~ основные функции, предоставляемые драйвером устройства (для открытия и закрытия устройства, чтения и записи данных, а также для поддержки любых других возможностей устройства, файловой системы или сети). Когда поступает запрос на операцию ввода-вывода, диспетчер ввода-вывода генерирует соответствующий IRP и вызывает нужный драйвер через одну из диспетчерских процедур этого драйвера.

dispatcher — диспетчер ядра ~ набор функций ядра, реализующих планирование в Windows . Единого модуля планировщика в Windows нет — соответствующий код распределен по всему ядру.

dispatcher database — база данных диспетчера ядра ~ набор структур данных, поддерживаемых ядром для принятия решений по планированию потоков. Эта база данных предназначена для учета того, какие потоки выполняются и на каких процессорах, а какие потоки ждут выполнения. *См. также* dispatcher ready queue.

dispatcher header — заголовок диспетчера ~ структура данных, содержащая тип объекта, его состояние (свободен/занят) и список потоков, ждущих на этом объекте.

dispatcher objects — объекты диспетчера ядра ~ набор объектов ядра, включающих средства синхронизации и способных влиять на планирование потоков. В число этих объектов входят мьютекс (его внутреннее название — мутант), событие, пара событий ядра, семафор, таймер и ожидаемый таймер.

dispatcher ready queue — очередь готовых (к выполнению) потоков, принадлежащая диспетчеру ядра ~ самая важная структура в базе данных диспетчера ядра (хранится в *KiDispatcherReadyListHead*). Эта очередь на самом деле представляет собой группу очередей — по одной для каждого уровня приоритета. Очереди содержат потоки, готовые к выполнению и ожидающие подключения к процессору.

display driver — драйвер видеоадаптера ~ драйвер, транслирующий аппаратно-независимые запросы на операции с графикой в запросы, специфичные для конкретного устройства. Далее запросы направляются соответствующему минипорт-драйверу режима ядра. Драйвер видеоадаптера отвечает за реализацию операций рисования, либо напрямую записывая данные в буфер кадров, либо взаимодействуя с чипом ускорителя графики на видеоплате.

driver object — объект «драйвер» ~ структура данных, которая представляет индивидуальный драйвер в системе и в которой регистрируется адрес каждой диспетчерской процедуры (входной точки) драйвера.

driver support routines — процедуры поддержки драйверов ~ процедуры, вызываемые драйверами устройств при выполнении запросов на ввод-вывод.

dynamic disk — динамический диск ~ диск, который поддерживает составные тома и более гибкую схему разбиения на разделы, чем у базовых дисков. *См. также* basic disk.

dynamic-link library (DLL) — динамически подключаемая библиотека (DLL) ~ набор функций, связанных в двоичный образ, который приложения могут динамически загружать при необходимости.

E

environment subsystems — подсистемы окружения ~ пользовательские процессы и DLL, которые предоставляют сервисы, встроенные в операционную систему, пользовательским приложениям, и тем самым формируют определенное окружение. Windows поддерживает две подсистемы окружения: Windows и POSIX. Windows NT 4.0 поддерживала и OS/2 1.2. Windows XP и более поздние версии поставляются только с подсистемой Windows, но вы можете бесплатно скачать продукт Microsoft Services for Unix, в состав которого входит расширенная версия подсистемы POSIX.

event — событие ~ объект, способный находиться в одном из двух состояний — свободном (signaled) или занятом (nonsignaled); применяется для синхронизации потоков. Событием также называется то, что заставляет систему выполнить определенное действие.

exception dispatcher — диспетчер исключений ~ модуль ядра, обслуживающий все исключения, кроме тех, которые могут быть разрешены обработчиком ловушки. Задача диспетчера исключений — найти обработчик исключений, способный «устранить» данное исключение.

executive — исполнительная система ~ верхний уровень Ntoskrnl.exe (ядро находится на более низком уровне). Исполнительная система предоставляет базовые сервисы операционной системы, например диспетчер процессов и потоков, диспетчер виртуальной памяти, диспетчер памяти, монитор состояния защиты, подсистему ввода-вывода и диспетчер кэша. *См. также* kernel.

executive objects — объекты исполнительной системы ~ объекты, реализуемые различными компонентами исполнительной системы (диспетчером процессов, диспетчером памяти, подсистемой ввода-вывода и т. д.). Эти объекты и их сервисы служат примитивами, из которых подсистемы окружения конструируют собственные версии объектов и другие ресурсы. Поскольку объекты исполнительной системы обычно создаются либо подсистемой окружения в интересах пользовательской программы, либо компонентами операционной системы, многие из них содержат (инкапсулируют) один или несколько объектов ядра. *См. также* kernel objects.

executive resources — ресурсы исполнительной системы ~ объекты, которые обеспечивают как монопольный доступ (например, мьютекс), так и разделяемый доступ для чтения (несколько потоков могут одновременно считывать данные из одной структуры). Эти объекты доступны только коду режима ядра — к ним нельзя обратиться через Windows API.

extended partition — дополнительный раздел ~ раздел особого типа, который содержит главную загрузочную запись (MBR) и свою таблицу разделов. Применение таких разделов позволяет операционным системам Microsoft создавать на диске более четырех разделов. *См. также* partition.

F

fast I/O — быстрый ввод-вывод ~ метод чтения и записи кэшируемого файла без генерации пакета запроса ввода-вывода (I/O request packet, IRP).

fast LPC — быстрый LPC ~ особый механизм межпроцессной связи, обеспечивающий обмен сообщениями между потоками.

file mapping object — объект «проекция файла» ~ объект Windows API, позволяющий использовать разделяемую память (его внутреннее название — объект-раздел). *См. также* section object.

file reference — файловая ссылка ~ 64-битное значение, которое идентифицирует файл на NTFS-томе. Файловая ссылка состоит из номера файла и номера последовательности. Первый номер соответствует позиции записи о данном файле в главной таблице файлов (MFT) за вычетом единицы (или позиции базовой записи о данном файле за вычетом единицы, если файл требует более одной записи в MFT).

file system driver (FSD) — драйвер файловой системы (FSD) ~ тип драйвера устройства режима ядра, который принимает запросы на ввод-вывод, обращенные к файлам, и на их основе выдает свои, более сложные запросы, адресуя их драйверам физических устройств. *См. также* local file system driver (FSD), network file system driver (FSD).

file system filter driver — драйвер фильтра файловой системы ~ тип драйвера устройства режима ядра, который перехватывает запросы ввода-вывода, дополнительно обрабатывает их и передает драйверам более низкого уровня.

file system format — формат файловой системы ~ определяет способ хранения файловых данных на носителях и характеристики файловой системы. Например, формат, не предусматривающий сопоставление прав доступа с файлами и каталогами, не позволяет файловой системе поддерживать защиту. Кроме того, формат файловой системы может накладывать ограничения на размеры файлов и емкости устройств внешней памяти.

filter driver — драйвер фильтра ~ *см.* file system filter driver.

foreground application — активное приложение ~ процесс, которому принадлежит поток, владеющий активным окном (находящимся в фокусе ввода).

free build — свободная (от отладочного кода) версия ~ версия Windows, которая может быть приобретена в системе розничной торговли. При ее сборке включаются все параметры оптимизации, поддерживаемые компилятором, и, кроме того, из всех образов удаляется информация о таблицах внутренних символов. *См. также* checked build.

function driver — функциональный драйвер ~ основной драйвер устройства, который предоставляет рабочий интерфейс к своему устройству. Обязателен, если только ввод-вывод на устройстве не выполняется драйвером шины и каким-либо драйвером фильтра шины, как в случае SCSI PassThru. Функциональный драйвер обычно является единственным, кто обращается к регистрам устройства.

G

granted access rights — предоставленные права доступа ~ права доступа, выданные потоку монитором состояния защиты в результате успешного открытия объекта.

Graphical Identification and Authentication (GINA) ~ DLL пользовательского режима, выполняемая в процессе Winlogon и применяемая Winlogon для получения имени и пароля пользователя или PIN-кода смарт-карты. Стандартная GINA — \Windows\System32\Msgina.dll.

H

handle — описатель ~ идентификатор объекта. Процесс получает описатель объекта при создании или открытии объекта по имени. Ссылка на объект по описателю действует быс-

трее, чем ссылка по имени, так как в первом случае диспетчеру объектов не приходится просматривать свое пространство имен.

handle table — таблица описателей ~ таблица, содержащая указатели на все объекты, описатели которых открыты процессом. Таблица описателей реализуется по трехуровневой схеме — по аналогии с тем, как аппаратный блок управления памятью в x86-системах реализует трансляцию виртуальных адресов в физические.

hardware abstraction layer (HAL) — уровень абстрагирования от оборудования (HAL) ~ загружаемый модуль режима ядра (Hal.dll), предоставляющий низкоуровневый интерфейс для аппаратной платформы, на которой работает Windows. HAL скрывает от операционной системы все, что связано с аппаратной реализацией функциональности, специфичной для архитектуры и конкретного оборудования, в том числе интерфейсы ввода-вывода, контроллеры прерываний и механизмы взаимодействия между процессорами.

hash — хэш ~ статистически уникальное значение, генерируемое из блока данных (например, из файла) с использованием алгоритмов шифрования. Поскольку разные данные дают разные хэши, их можно использовать для распознавания изменений в данных в процессе передачи (скажем, попыток взлома или умышленной модификации информации). Хэши используются механизмом поддержки цифровых подписей драйверов Windows.

heap — куча (динамически распределяемая память) ~ область, которая состоит из одной или нескольких страниц памяти и из которой функции диспетчера куч позволяют выделять блоки памяти меньшего размера.

heap manager — диспетчер куч ~ набор функций, позволяющих выделять и освобождать блоки памяти различного размера (который может быть меньше размера страниц памяти). Эти функции находятся в двух местах — в Ntdll.dll и Ntoskrnl.exe. API подсистемы окружения используют функции диспетчера куч, размещенные в Ntdll, а компоненты исполнительной системы и драйверы устройств — функции, содержащиеся в Ntoskrnl.

hive — куст ~ один из нескольких файлов на диске, содержащих информацию реестра. Каждый куст хранит дерево реестра, где один из разделов выступает в роли корня данного дерева.

hyperspace — гиперпространство ~ особый регион, на который проецируется список рабочего набора процесса. Кроме того, он используется для временного проецирования физических страниц при таких операциях, как обнуление страницы из списка свободных страниц (когда список обнуленных страниц пуст и возникает потребность в обнуленной странице), объявление недействительными PTE в других таблицах страниц (когда страница удаляется из списка простаивающих страниц) и создание адресного пространства дочернего процесса.

I

I/O completion routine — процедура завершения ввода-вывода ~ процедура, реализуемая для уведомления драйвера более высокого уровня о том, что драйвер более низкого уровня закончил обработку IRP (пакета запроса ввода-вывода). Например, диспетчер ввода-вывода вызывает такую процедуру драйвера файловой системы после того, как драйвер устройства завершает передачу данных в файл или из файла. Процедура завершения уведомляет драйвер файловой системы об успехе, неудаче или отмене операции и позволяет ему выполнить соответствующие действия.

I/O request packet (IRP) — пакет запроса ввода-вывода (IRP) ~ структура данных, контролирующая обработку операции ввода-вывода на каждом этапе. Большинство запросов на ввод-вывод представляется IRP, которые передаются от одного компонента подсистемы ввода-вывода другому.

I/O system — подсистема ввода-вывода ~ компонент исполнительной системы, который принимает запросы на ввод-вывод (поступающие от потоков как пользовательского режима, так и режима ядра) и, преобразовав в нужный формат, доставляет их соответствующим устройствам ввода-вывода.

ideal processor — идеальный процессор ~ предпочтительный процессор для определенного потока.

idle summary — сводка простоя ~ битовая маска (*KiIdleSummary*), в которой каждый установленный бит представляет простаивающий процессор.

impersonation — олицетворение, подмена ~ функциональность, позволяющая закреплять за потоками маркеры доступа, отличные от тех, которые назначены процессу.

initialization routine — процедура инициализации ~ процедура драйвера, выполняемая диспетчером ввода-вывода при загрузке драйвера в операционную систему. Эта процедура создает системные объекты, используемые диспетчером ввода-вывода для распознавания драйвера и обращения к нему.

in-paging I/O — ввод-вывод, связанный с подкачкой страниц в память ~ ввод-вывод, вызываемый ошибкой страницы и требующий операции чтения из файла (страничного или проецируемого) для загрузки соответствующей страницы в память. Такой ввод-вывод является синхронным (поток ждет его окончания на каком-либо событии) и не может быть прерван доставкой APC.

instancing — создание экземпляра ~ здесь: термин, относящийся к созданию отдельных копий одних и тех же частей пространства имен. Создание экземпляров \DosDevices позволяет каждому пользователю присваивать одним и тем же дискам разные буквы и получать свой набор Windows-объектов вроде последовательных портов.

intelligent file read-ahead — интеллектуальное опережающее чтение файловых данных ~ метод предсказания того, какие данные будут скорее всего считаны вызывающим потоком в следующий раз.

inter-processor interrupt (IPI) — межпроцессорное прерывание ~ прерывание, инициируемое ядром, чтобы запросить выполнение какой-либо операции другим процессором, например подключение определенного потока к этому процессору или обновление его TLB-кэша.

interrupt — прерывание ~ асинхронное событие (которое может произойти в любой момент), не связанное с текущим кодом, выполняемым процессором. Прерывания генерируются в основном устройствами ввода-вывода и системными таймерами.

interrupt dispatch table (IDT) — таблица диспетчеризации прерываний (IDT) ~ структура данных, используемая Windows для поиска процедуры, способной обработать конкретное прерывание. Уровень запроса прерывания (IRQL) источника прерывания используется как индекс таблицы, а соответствующий элемент в таблице указывает на искомую процедуру обработки прерывания.

interrupt dispatcher — диспетчер прерываний ~ часть обработчика ловушек, которая реагирует на прерывания.

interrupt object — объект прерывания ~ управляющий объект ядра, который позволяет драйверам устройств регистрировать процедуры обслуживания прерываний (ISR) для своих устройств. Объект прерывания содержит всю информацию, необходимую ядру для того, чтобы сопоставить ISR устройства с конкретным уровнем прерывания. В эту информацию включается адрес ISR, IRQL, при котором возникает прерывание от устройства, и элемент в таблице диспетчеризации прерываний, с которым следует связать эту ISR.

interrupt request (IRQ) — запрос прерывания (IRQ) ~ значение, идентифицирующее прерывание. В x86-системах внешние прерывания ввода-вывода поступают по одной из линий контроллеру прерываний. Контроллер в свою очередь прерывает работу процессора (по единственной линии). Как только процессор прерывается, он запрашивает контроллер получить IRQ. Контроллер прерываний преобразует IRQ в номер прерывания, который используется как индекс в таблице диспетчеризации прерываний (IDT), и передает управление соответствующей процедуре. При загрузке системы Windows заполняет IDT указателями на все процедуры ядра, обрабатывающие прерывания и исключения.

interrupt request level (IRQL) — уровень запроса прерывания (IRQL) ~ схема приоритетов прерываний в Windows. Ядро нумерует уровни IRQL от 0 до 31 (или от 0 до 15 в 64-разрядной Windows), причем бóльшие числа соответствуют более высоким приоритетам. Хотя ядро определяет стандартный набор IRQL для программных прерываний, HAL также увязывает номера аппаратных прерываний с уровнями IRQL.

interrupt service routine (ISR) — процедура обслуживания прерываний (ISR) ~ процедура драйвера устройства, которой диспетчер прерываний, принадлежащий ядру, передает управление при прерывании от устройства. В модели ввода-вывода Windows ISR работают при высоком уровне IRQL, поэтому они выполняют минимум команд, чтобы избежать излишней блокировки прерываний более низкого уровня. ISR ставит в очередь DPC, которая выполняется при более низком IRQL и отвечает за основную часть обработки прерывания. ISR могут быть только у драйверов устройств.

J

job object — объект «задание» ~ это именуемый, защищаемый и разделяемый объект ядра, обеспечивающий управление одним или несколькими процессами как группой. Объект «задание» также регистрирует базовую учетную информацию всех включенных в него процессов, в том числе уже завершившихся.

journaling — ведение журнала ~ метод протоколирования, изначально разработанный для обработки транзакций. Используется восстанавливаемыми файловыми системами вроде NTFS для поддержания целостного состояния тома.

K

kernel — ядро ~ самый нижний уровень в Ntoskrnl.exe. Ядро, компонент исполнительной системы, определяет, как операционная система использует процессор или процессоры, и гарантирует, что они используются эффективно. Ядро отвечает за планирование и диспетчеризацию потоков, обработку ловушек и диспетчеризацию исключений, обработку и диспетчеризацию прерываний, а также за синхронизацию процессоров в многопроцессорных системах. *См. также* executive.

kernel debugger — отладчик ядра ~ инструмент, применяемый для отладки драйверов, выявления причины краха операционной системы и анализа аварийных дампов. Также полезен как инструмент для исследования внутреннего устройства Windows, поскольку он сообщает системную информацию, недоступную стандартным утилитам.

kernel handle table — таблица описателей, принадлежащая ядру ~ таблица (с внутренним именем *ObpKernelHandleTable*), описатели в которой доступны только в режиме ядра в контексте любого процесса. Функции режима ядра могут ссылаться на описатели в этой таблице из контекста любого процесса без отрицательных последствий для общей производительности системы.

kernel memory dump — дамп памяти ядра ~ тип дампа памяти (в системах Windows Server предлагается по умолчанию), который включает лишь страницы (как для чтения, так и для записи) режима ядра, находящиеся в физической памяти на момент краха. Страницы, принадлежащие пользовательским процессам, не включаются. Поскольку только код режима ядра может напрямую вызывать крах Windows, содержимое страниц пользовательских процессов обычно ничего не дает для понимания причин краха. Заранее предсказать объем дампа памяти ядра нельзя, так как он зависит от объема памяти ядра, выделенной операционной системой и драйверами.

kernel mode — режим ядра ~ привилегированный режим выполнения кода, при котором доступна вся память и можно выполнять любые команды процессора. В этом режиме работает код операционной системы. *См. также* user mode.

kernel objects — объекты ядра ~ набор примитивных объектов, реализуемых ядром Windows. Эти объекты невидимы коду пользовательского режима — они создаются и используются только внутри исполнительной системы. Объекты ядра предоставляют базовую функциональность вроде синхронизации, на которую опираются объекты исполнительной системы. *См. также* executive objects.

kernel streaming filter drivers — драйверы потоковых фильтров ядра ~ драйверы режима ядра, соединенные в цепочку для обработки потоковых данных, например при записи или воспроизведении аудио- и видеоинформации.

kernel-mode device driver — драйвер устройства режима ядра ~ единственный тип драйверов, способных напрямую обращаться к аппаратным устройствам и управлять ими.

kernel-mode graphics driver — графический драйвер режима ядра ~ драйвер видеоадаптера или принтера подсистемы Windows, который преобразует аппаратно-независимые GDI-запросы в запросы, специфичные для конкретных устройств.

key — раздел ~ механизм ссылок на данные в реестре. С объектом «раздел реестра» сопоставляется произвольное количество параметров, содержащих собственно данные.

key control block — блок управления разделом ~ структура, в которой хранится полное имя раздела реестра, индекс ячейки раздела, на которую ссылается данный блок управления, и флаг, указывающий, надо ли диспетчеру конфигурации удалять ячейку раздела (на которую ссылается данный блок) при за-крытии последнего описателя этого раздела. В Windows все блоки управления разделами помещаются в двоичное дерево с сортировкой по алфавиту, что позволяет быстро находить нужный блок по имени.

key object — объект «раздел реестра» ~ тип объекта, определяемый диспетчером конфигурации для интеграции пространства имен реестра с универсальным пространством имен ядра.

keyed event — событие с ключом ~ синхронизирующий объект, введенный в Windows XP.

L

last known good control set — набор параметров последней удачной конфигурации ~ копия критически важной для загрузки системы информации в разделе реестра HKLM\SYSTEM\CurrentControlSet, создаваемая после успешного входа пользователя в систему. Этот набор параметров можно выбрать в стартовом меню, если какие-то изменения в конфигурации не дают возможности загрузить систему.

lazy writer — подсистема отложенной записи ~ набор системных потоков, вызывающих диспетчер памяти для сброса содержимого кэша на диск в фоновом режиме (асинхронная запись на диск). Благодаря этой подсистеме диспетчер кэша оптимизирует дисковый ввод-вывод.

legacy drivers — унаследованные драйверы ~ драйверы устройств, написанные для Microsoft Windows NT, но загружаемые в Windows без всяких изменений в исходном коде. Такие драйверы не поддерживают управление электропитанием и не взаимодействуют с диспетчером Plug and Play.

local file system driver (FSD) — локальный драйвер файловой системы ~ драйвер, управляющий локальными томами компьютера. *См. также* file system driver.

local kernel debugging — локальная отладка ядра ~ возможность подключения отладчика ядра к локальной работающей системе для просмотра ее внутреннего состояния (в противоположность подключению к целевой системе, загруженной в отладочном режиме, с использованием нуль-модемного кабеля или кабеля 1394).

local procedure call (LPC) ~ механизм межпроцессной связи для высокоскоростного обмена сообщениями. Этот внутренний механизм не доступен через Windows API — к нему могут обращаться только компоненты операционной системы Windows. LPC обычно используется для взаимодействия серверного процесса с одним или несколькими клиентскими процессами. LPC-соединение может быть установлено между двумя процессами пользовательского режима или между процессом пользовательского режима и компонентом режима ядра.

Local Security Authentication Subsystem (LSASS) — подсистема локальной аутентификации (LSASS) ~ системный процесс пользовательского режима, отвечающий за аутентификацию учетных записей, по которым пользователи или приложения пытаются обратиться к данному компьютеру.

Local System account — учетная запись локальной системы ~ предопределенная локальная учетная запись, используемая для запуска службы и формирования контекста защиты для этой службы. Эта учетная запись называется NT AUTHORITY\System. В ней не используется пароль, и любой заданный вами пароль будет проигнорирован. Эта запись предоставляет полный доступ к системе. Поскольку учетная запись локальной системы действует как компьютер в сети, она получает доступ к сетевым ресурсам.

log file — файл журнала ~ файл метаданных (с именем $LogFile), используемый NTFS для регистрации всех операций, влияющих на структуру NTFS-тома. Этот файл применяется для восстановления NTFS-тома после аварии системы.

log hive — регистрационный куст ~ куст, используемый диспетчером конфигурации для поддержки восстановления неизменяемого куста реестра (связанного с дисковым файлом). С каждым неизменяемым кустом сопоставлен свой регистрационный куст — скрытый файл с тем же базовым именем, но с расширением LOG. *См. также* hive.

logging — протоколирование ~ метод обработки транзакций, применяемый NTFS для поддержания целостности файловой системы на случай каких-либо сбоев. При этом подоперации любой транзакции, меняющей важные структуры данных файловой системы, регистрируются в файле журнала еще до их выполнения, поэтому после сбоя частично выполненную транзакцию можно повторить или отменить.

logical cluster numbers (LCN) — логические номера кластеров (LCN) ~ номера кластеров от начала тома и до конца; по ним NTFS ссылается на физические участки диска. Для преобразования LCN в физический адрес на диске NTFS умножает LCN на кластерный множитель (cluster factor), получая физическое смещение на диске в байтах, как того требует интерфейс драйвера диска.

logical prefetcher — средство логической предвыборки ~ компонент ядра, который отслеживает обращения к файлам при загрузке системы и приложений и считывает соответствующие данные для ускорения последующих процессов загрузки.

logical sequence numbers (LSNs) — номера логической последовательности (LSN) ~ номера, используемые NTFS для идентификации записей, вносимых в файл журнала.

logon process — процесс входа ~ процесс пользовательского режима, выполняющий Winlogon.exe — сервис, который отвечает за прием имени и пароля пользователя, передачу этой информации серверу локальной аутентификации для проверки и за создание начального процесса в сеансе пользователя.

look-aside list — ассоциативный список ~ высокоскоростной механизм выделения памяти блоками фиксированного размера. Ассоциативные списки могут быть подкачиваемыми или неподкачиваемыми — в зависимости от того, в каком пуле они создаются.

M

mapped file I/O — ввод-вывод в проецируемые файлы ~ ввод-вывод в дисковый файл, являющийся частью виртуальной памяти процесса. Программа может обращаться к такому файлу как к большому массиву без буферизации данных или выполнения дискового ввода-вывода. Все операции, связанные с вводом-выводом, берет на себя диспетчер памяти, который использует для этого механизм подкачки.

master file table (MFT) — главная таблица файлов (MFT) ~ центральное звено структуры NTFS-тома. MFT реализуется как массив записей о файлах. Размер каждой записи фиксирован и составляет 1 Кб независимо от размера кластеров.

memory manager — диспетчер памяти ~ компонент исполнительной системы, который реализует виртуальную память с подкачкой по требованию и создает иллюзию того, что у каждого процесса имеется свое большое виртуальное адресное пространство (на физическую память проецируется лишь некое подмножество этого пространства).

metadata — метаданные ~ данные, описывающие файлы на диске; также называются данными о структуре тома.

metadata files — файлы метаданных ~ набор файлов на каждом NTFS-томе, которые хранят информацию, используемую для реализации структуры файловой системы.

MFT mirror — зеркальная копия MFT ~ файл метаданных NTFS (с именем $MFTMirr), размещаемый в середине диска и содержащий первые несколько строк из главной таблицы файлов.

miniport driver — минипорт-драйвер ~ разновидность драйвера устройства режима ядра; преобразует универсальный запрос ввода-вывода в специфический для конкретного типа адаптеров, например SCSI-адаптеров.

mirrored volume — зеркальный том ~ том, содержимое которого дублируется на равном разделе другого диска. Зеркальные тома иногда называются RAID уровня 1 (RAID-1).

modified page writer — подсистема записи модифицированных страниц ~ два системных потока в диспетчере виртуальной памяти, отвечающие за ограничение размера списка модифицированных страниц. При чрезмерном увеличении этого списка страницы записываются в соответствующие места — в страничный файл (поток *MiModifiedPageWriter*) или проецируемые файлы (поток *MiMappedPageWriter*).

mount — монтирование ~ метод, применяемый NTFS при первом обращении к тому; в этом контексте монтирование подразумевает подготовку тома к использованию. Чтобы смонтировать том, NTFS просматривает загрузочный файл и ищет физический адрес (на диске) главной таблицы файлов.

mount points — точки монтирования ~ механизм, позволяющий связывать тома через каталоги на NTFS-томах, что позволяет обращаться к томам, которым не назначена буква диска. Точки монтирования поддерживаются точками повторного разбора.

MSDN ~ Microsoft Developer Network, программа Microsoft для поддержки разработчиков. MSDN предлагает три уровня подписки: MSDN Library, Professional и Universal. Более подробную информацию см. на *msdn.microsoft.com*.

multipartition volume — составной том ~ объект, который представляет несколько разделов и позволяет файловой системе управлять этими разделами как единым целым. Составные тома обеспечивают такие уровни производительности, надежности и масштабировании, которые невозможны при использовании простых томов.

mutant — мутант ~ внутреннее название мьютекса.

mutex — мьютекс ~ синхронизирующий механизм, используемый для упорядочения доступа к общему ресурсу.

N

native application — встроенное приложение ~ приложение, использующее только системные сервисы Ntdll и не являющееся клиентом подсистемы Windows. Пример такого приложения — Smss (Session Manager).

network file system driver (FSD) — сетевой драйвер файловой системы ~ драйвер, который дает возможность пользователям обращаться к данным на томах удаленных компьютеров. *См. также* file system driver.

network logon service — служба сетевого входа ~ сервис пользовательского режима в процессе Services.exe, реагирующий на запросы сетевого входа. Аутентификация обрабатывается как при локальном входе, и соответствующая информация передается LSASS для проверки.

network redirector and server — сетевые редиректор и сервер ~ драйверы файловой системы, первый из которых передает запросы удаленного ввода-вывода соответствующей машине в сети, а второй — принимает такие запросы.

nonpaged pool — пул неподкачиваемой памяти ~ пул памяти, состоящий из диапазонов системных виртуальных адресов, которые всегда находятся в физической памяти и поэтому доступны из любого адресного пространства без обращения к механизму подкачки. Такой пул создается при инициализации системы и используется компонентами режима ядра для выделения системной памяти.

Ntdll.dll ~ особая библиотека системной поддержки, которая используется в основном DLL-модулями подсистем и которая содержит диспетчерские интерфейсы к системным сервисам исполнительной системы и внутренним функциям поддержки.

Ntkrnlmp.exe ~ исполнительная система и ядро для многопроцессорных систем.

Ntoskrnl.exe ~ исполнительная система и ядро для однопроцессорных систем.

O

object — объект ~ в исполнительной системе Windows: единственный экземпляр периода выполнения (run-time instance) статически определенного типа объекта.

object attribute — атрибут объекта ~ поле данных в объекте, которое частично определяет состояние этого объекта.

object directory — каталог объектов ~ контейнер других объектов. Каталог объектов позволяет реализовать иерархическое пространство имен, внутри которого хранятся объекты других типов.

object handle — описатель объекта ~ индекс в таблице описателей, специфичный для конкретного процесса; на эту таблицу указывает блок EPROCESS.

object manager — диспетчер объектов ~ компонент исполнительной системы, отвечающий за создание, удаление, защиту и учет объектов. Диспетчер объектов обеспечивает централизованное управление ресурсами.

object methods — методы объекта ~ средства манипулирования объектами — обычно для чтения или изменения их атрибутов. Например, в случае процесса метод *open* принимает идентификатор процесса и возвращает указатель на объект «процесс».

object reuse protection — защита от повторного использования объектов ~ средства, предотвращающие доступ пользователей к данным, удаленным другим пользователем, или к освобожденной им памяти. Такая защита ликвидирует потенциальные бреши в защите, инициализируя все объекты, файлы и области памяти перед выделением их какому-либо пользователю.

object type — тип объекта ~ тип данных, определенных в системе и включающий сервисы, которые оперируют с экземплярами этого типа данных, и набор атрибутов объекта; иногда называется классом объекта.

P

page directory — каталог страниц ~ страница, создаваемая диспетчером памяти для хранения адресов всех таблиц страниц процесса. В каждом процессе имеется один каталог страниц.

page directory entry (PDE) — элемент каталога страниц (PDE) ~ каталог страниц состоит из PDE (размером по 4 байта), которые описывают состояние и адреса всех таблиц страниц, возможных для процесса.

page fault — ошибка страницы ~ попытка обращения к недействительной странице (отсутствующей в физической памяти). Обработчик ловушек, принадлежащий ядру, пересылает такие ошибки соответствующему обработчику диспетчера памяти (функции *MmAccessFault*).

page file backed section — раздел, поддерживаемый страничным файлом ~ объект «раздел», проецируемый на переданную память.

page frame database — база данных фреймов страниц ~ база данных, описывающая состояние каждой страницы физической памяти. Страницы могут находиться в одном из восьми состояний: активная (или действительная), в переходном состоянии, простаивающая, модифицированная, модифицированная, но не записываемая, свободная, обнуленная или аварийная.

page frame number (PFN) database — база данных номеров фреймов страниц (PFN) ~ *см.* page frame database.

page table — таблица страниц ~ страница с информацией о проецировании (состоит из массива PTE). Операционная система создает такую таблицу для описания адресов виртуальных страниц в адресном пространстве процесса. Поскольку адресные пространства являются закрытыми, у каждого процесса имеется свой набор таблиц страниц. Таблицы страниц, описывающие системное пространство, являются общими для всех процессов.

page table entry (PTE) — элемент таблицы страниц (PTE) ~ элемент в таблице страниц, содержащий адрес, на который проецируется виртуальный адрес. Эта таблица может находиться как в физической памяти, так и в страничном файле на диске.

paged pool — пул подкачиваемой памяти ~ регион виртуальной памяти в системном пространстве, который может подгружаться в рабочий набор системного процесса и выгружаться из него. Пул подкачиваемой памяти создается при инициализации системы и используется компонентами режима ядра для выделения системной памяти. В однопроцессорных системах таких пулов два, а в многопроцессорных — четыре.

paging — подкачка ~ перемещение содержимого памяти на диск с целью освобождения физической памяти для использования другими процессами или самой системой, а также загрузка страниц с диска в физическую память.

partition — раздел (дисковый) ~ область жесткого диска в операционных системах Microsoft. Файловые системы (вроде FAT и NTFS) форматируют каждый раздел как том. На жестком диске может быть до четырех главных разделов. *См. также* extended partition.

partition table — таблица разделов ~ часть главной загрузочной записи (MBR), состоящая из четырех элементов, каждый из которых определяет местонахождение одного из четырех главных разделов на диске. В таблицу разделов также записывается тип раздела; последний определяет файловую систему на этом разделе.

Physical Address Extension (PAE) ~ режим проецирования памяти, поддерживаемый всеми процессорами типа Intel x86, начиная с Pentium Pro. При наличии соответствующего чипсета режим PAE позволяет адресоваться к 64 Гб физической памяти. Когда процессор работает в режиме PAE, блок управления памятью (MMU) разбивает виртуальные адреса на четыре поля.

Plug and Play (PnP) manager — диспетчер Plug and Play (PnP) ~ один из основных компонентов исполнительной системы, который определяет и загружает драйверы, нужные для поддержки конкретного устройства. Диспетчер PnP выясняет требования каждого устройства к аппаратным ресурсам в процессе перечисления. Исходя из требований к ресурсам, диспетчер PnP выделяет каждому устройству соответствующие аппаратные ресурсы — порты ввода-вывода, IRQ-линии, DMA-каналы и диапазоны памяти. Он также отвечает за рассылку уведомлений о любых изменениях в аппаратной конфигурации системы.

port driver — порт-драйвер ~ разновидность драйвера устройства режима ядра. Реализует обработку запроса на ввод-вывод, специфичного для конкретного типа порта ввода-вывода, например для SCSI.

power manager — диспетчер электропитания ~ один из основных компонентов исполнительной системы, который координирует события, связанные с управлением электропитанием, и генерирует соответствующие уведомления для драйверов устройств. Когда система простаивает, диспетчер электропитания может переводить ее в состояние пониженного энергопотребления. Изменения в уровне энергопотребления реализуются драйверами конкретных устройств, но координируются диспетчером электропитания.

printer driver — драйвер принтера ~ драйвер, транслирующий аппаратно-независимые запросы на графические операции в команды, специфичные для принтера. Эти команды обычно пересылаются порт-драйверу режима ядра — драйверу параллельного порта (Parport.sys) или USB-порта принтера (Usbprint.sys).

private cache map — закрытая карта кэша ~ структура, хранящая последние два адреса, по которым считывались данные. Эта информация помогает диспетчеру кэша выполнять интеллектуальное опережающее чтение данных.

process — процесс ~ виртуальное адресное пространство и управляющая информация, необходимая для выполнения набора объектов-потоков.

process ID — идентификатор процесса ~ уникальный идентификатор процесса (его внутреннее название — идентификатор клиента).

process working set — рабочий набор процесса ~ подмножество виртуального адресного пространства процесса, резидентное в физической памяти и принадлежащее выполняемому процессу. *См. также* system working set.

processor affinity — привязка к процессорам ~ набор процессоров, на которых может выполняться данный поток.

processor control register (PCR) ~ структура данных и ее расширение — PRCB (processor control block) — содержат информацию о состоянии каждого процессора в системе, в том числе текущий IRQL, указатель на аппаратную IDT, сведения о текущем потоке и потоке, который будет выполняться следующим. Ядро и HAL используют эту информацию для выполнения операций, специфичных для данной машины и ее архитектуры. Отдельные части структур PCR и PRCB документированы и определены в заголовочном файле Ntddk.h (в Windows DDK).

program — программа ~ статическая последовательность инструкций.

protected-mode — защищенный режим ~ состояние на одном из этапов за-грузки системы, при котором трансляция виртуальных адресов в физические еще не поддерживает-

ся, но становится доступной 32-разрядная адресация к памяти. После того как система переходит в защищенный режим, Ntldr может обращаться ко всей физической памяти.

protocol driver — драйвер протокола ~ драйвер, реализующий сетевой протокол.

prototype page table entry (prototype PTE) — прототипный элемент таблицы страниц (PTE) ~ программная структура, используемая диспетчером памяти для проецирования потенциально разделяемых страниц. Массив прототипных PTE создается при первом создании объекта «раздел».

Q

Quality of Service (QoS) ~ сетевая технология, гарантирующая выделение согласованных полос пропускания и поддержку ряда других параметров.

quantum — квант ~ отрезок времени, в течение которого поток может работать до того, как Windows прервет его выполнение, чтобы проверить, не ожидает ли выполнения другой поток с тем же уровнем приоритета.

quantum unit — квантовая единица ~ значение, которое отражает, сколько времени поток может выполняться до истечения его кванта. Это значение выражается не в единицах времени, а целым числом.

queued spinlock — спин-блокировка с очередью ~ спин-блокировка особого типа, используемая исключительно ядром и не экспортируемая компонентам исполнительной системы или драйверам устройств. Спин-блокировка с очередью обеспечивает более эффективную работу многопроцессорных систем, чем стандартная. *См. также* spinlock.

R

RAID-5 volume — том RAID-5 ~ отказоустойчивый вариант обычного чередующегося тома. Отказоустойчивость достигается за счет выделения дополнительного диска для хранения информации о четности для каждой чередующейся области (stripe).

real-mode — реальный режим ~ режим, в котором трансляция виртуальных адресов в физические не поддерживается и программам доступен лишь первый мегабайт физической памяти. В этом режиме выполняются только простые программы MS-DOS.

recoverability — восстанавливаемость ~ дополнительная возможность NTFS, позволяющая восстанавливать структуру тома после сбоя системы. В этом случае метаданные тома могут оказаться в рассогласованном состоянии, что приведет к повреждению большого числа файлов и каталогов. NTFS реализует восстанавливаемость за счет протоколирования изменений метаданных как транзакций, так что структуры файловой системы могут быть возвращены в согласованное состояние без потери информации о структуре каталогов и файлов. (Однако файловые данные могут быть потеряны.)

redo information — информация для повтора ~ информация, включаемая в запись контрольной точки NTFS и указывающая, как повторить подоперацию полностью запротоколированной транзакции, если сбой системы произошел перед самым сбросом данных транзакции из кэша на диск.

reference count — счетчик ссылок ~ счетчик, поддерживаемый диспетчером объектов и сообщающий, сколько указателей на объект выдано системным процессам. Диспетчер объектов увеличивает счетчик ссылок на 1 при каждой выдаче указателя на данный объект; когда компоненты режима ядра заканчивают использование указателя, они вызывают диспетчер объектов для уменьшения счетчика ссылок на соответствующий объект.

reparse data — данные повторного разбора ~ пользовательские данные, сопоставленные с файлом или каталогом, которые могут быть считаны из точки повторного разбора приложением, создавшим эти данные, драйвером фильтра файловой системы или диспетчером ввода-вывода.

reparse point — точка повторного разбора ~ NTFS-файл или каталог, с которым сопоставлен блок данных, называемый данными повторного разбора.

resident attribute — резидентный атрибут ~ атрибут, хранящийся непосредственно в главной таблице файлов. (Если файл невелик, все его атрибуты и их значения хранятся в записи этой таблицы.)

resource arbitration — арбитраж ресурсов ~ процесс, в ходе которого диспетчер Plug and Play (PnP) распределяет ресурсы между устройствами с учетом их требований. Поскольку устройства могут быть добавлены в систему после распределения ресурсов на этапе загрузки, диспетчер PnP должен уметь перераспределять ресурсы.

restricted token — ограниченный маркер ~ маркер, созданный из основного маркера или из маркера олицетворения с помощью функции *CreateRestrictedToken*. Ограниченный маркер является копией маркера, из которого он создан, но в него может быть внесен ряд изменений, например удалены какие-либо привилегии.

ring — кольцо ~ уровень привилегий, определенный в архитектурах процессоров x86 и x64 для защиты системного кода и данных от случайной или намеренной перезаписи кодом с меньшими привилегиями. Windows на платформах x86 и x64 использует уровень привилегий 0 (кольцо 0) в режиме ядра и 3 (кольцо 3) в пользовательском режиме.

S

safe mode — безопасный режим ~ конфигурация системы с минимальным набором драйверов устройств и сервисов. При этом драйверы и сервисы от сторонних поставщиков не загружаются.

SAM database — база данных SAM ~ база данных (в разделе реестра HKLM\ SAM), в которой хранятся определения пользователей и групп, а также их пароли и другие атрибуты.

scatter/gather I/O — ввод-вывод по механизму «scatter/gather» ~ разновидность высокопроизводительного ввода-вывода, поддерживаемая Windows и доступная через Windows-функции *ReadFileScatter* и *WriteFileGather*. Эти функции позволяют в рамках одной операции считывать или записывать данные из нескольких буферов в виртуальной памяти в непрерывную область дискового файла. Чтобы задействовать такой ввод-вывод, нужно открыть файл для некэшируемого асинхронного (перекрывающегося) ввода-вывода и выровнять пользовательские буферы по границам страниц.

section object — объект «раздел» ~ объект, представляющий блок памяти, до-ступный двум и более процессам для совместного использования. Объект-раздел можно проецировать на страничный файл или другой файл на диске. Исполнительная система использует разделы для загрузки исполняемых образов в память, а диспетчер кэша — для доступа к данным в кэшированном файле. В подсистеме Windows такой объект называется проекцией файла.

section object pointers — указатели объекта «раздел» ~ структура для каждого открытого файла (представленного объектом «файл»), играющая ключевую роль в поддержании целостности данных при всех типах доступа к файлу и в обеспечении кэширования файлов. Эта структура указывает на одну или две области управления: одна из них используется для проецирования файла при доступе к нему как к файлу данных, а другая — при его выполнении как исполняемого образа.

sector — сектор ~ аппаратно адресуемый блок физического диска. Размер секторов на жестких дисках в x86-системах почти всегда равен 512 байтам.

secure attention sequence (SAS) ~ комбинация клавиш, нажатие которой уведомляет Winlogon о запросе пользователя на вход в систему.

Security Accounts Manager (SAM) service — Служба диспетчера учетных записей безопасности (SAM) ~ набор подпрограмм, отвечающих за управление базой данных, которая содержит имена пользователей и групп, определенных на локальной машине или в домене (если данная система является контроллером домена). SAM выполняется в контексте LSASS.

security auditing — аудит безопасности ~ механизм, позволяющий Windows регистрировать события, связанные с защитой, а также любые попытки создания, удаления и обращения к системным ресурсам. При входе регистрируются идентификационные данные всех пользователей, что дает возможность легко выявить любого, кто попытался выполнить несанкционированную операцию.

security descriptor — дескриптор защиты ~ структура данных, указывающая, кто может выполнять операции над объектом и какие именно операции. Де-скриптор защиты состоит из атрибутов.

security identifier (SID) — идентификатор защиты (SID) ~ средство уникальной идентификации сущностей, выполняющих какие-либо операции в системе. SID представляет собой числовое значение переменной длины, формируемое из номера версии структуры SID, 48-битного кода агента идентификатора (identifier authority value) и переменного количества 32-битных кодов суб-агентов и/или относительных идентификаторов (relative identifiers, RID).

Security Reference Monitor (SRM) — монитор состояния защиты (SRM) ~ компонент исполнительной системы (Ntoskrnl.exe), который вводит в действие политику безопасности на локальном компьютере. Он охраняет ресурсы операционной системы, обеспечивая защиту и аудит объектов периода выполнения.

semaphore — семафор ~ счетчик, позволяющий обращаться к ресурсу некоему максимальному числу потоков.

server process — серверный процесс ~ пользовательский процесс, выполняемый как сервис Windows, например Event Log и Schedule. Многие серверные приложения вроде Microsoft SQL Server и Microsoft Exchange Server включают компоненты, выполняемые как сервисы Windows.

session — сеанс ~ состоит из процессов и других системных объектов (вроде WindowStation, рабочих столов и окон). Эти объекты представляют сеанс единственного пользователя, который зарегистрировался на рабочей станции. У каждого сеанса есть своя область пула подкачиваемой памяти, используемая Win32k.sys для выделения памяти под закрытые GUI-структуры данных сеанса. Кроме того, каждый сеанс получает свою копию процесса подсистемы Windows (Csrss.exe) и Winlogon.exe.

session space — пространство сеанса ~ часть системного пространства, на которую проецируются все общие для сеанса структуры данных. Список рабочего набора сеанса описывает части пространства сеанса, резидентные в памяти.

shared cache map — общая карта кэша ~ структура, описывающая состояние кэшируемого файла, включая его размер и (из соображений безопасности) длину действительных данных.

shared memory — разделяемая (общая, совместно используемая) память ~ память, видимая более чем одному процессу или присутствующая более чем в одном виртуальном адресном пространстве.

signal state — свободное состояние ~ состояние синхронизирующего объекта.

simple volume — простой том ~ набор объектов, который представляет секторы единственного раздела, управляемые драйвером файловой системы как единым целым.

single sign-on — унифицированный вход в систему ~ возможность проверки регистрационной информации сразу на нескольких системах. Например, пользователь, входящий в систему Windows, может быть одновременно аутентифицирован на UNIX-сервере.

small memory dump — малый дамп памяти ~ тип дампа памяти (вариант по умолчанию в системах Windows Professional) объемом 64 Кб. Он включает в себя стоп-код с параметрами, список загруженных драйверов устройств, структуры данных, описывающие текущие процесс и поток (EPROCESS и ETHREAD), а также стек ядра для вызвавшего крах потока.

spanned volume — перекрытый том ~ единый логический том, состоящий максимум из 32 разделов на одном или нескольких дисках. Отдельные разделы объединяются в перекрытый том (с помощью оснастки Disk Management консоли Microsoft Management Console), который после этого можно отформатировать под любую файловую систему, поддерживаемую Windows.

sparse file — разреженный файл ~ файлы, часто очень большие, но содержащие лишь малые объемы ненулевых данных.

spinlock — спин-блокировка ~ механизм взаимоисключающей блокировки, используемый ядром. *См. также* queued spinlock.

stream — поток данных ~ последовательность байтов в файле.

striped volume — чередующийся том ~ набор разделов (максимум 32, по одному разделу на одном диске), объединенных в единый логический том. Чередующиеся тома также

называются томами RAID уровня 0 (RAID-0). Раздел чередующегося тома не обязательно должен охватывать целый диск; единственное ограничение — все разделы должны быть равного размера.

stop code — стоп-код ~ код, который классифицирует тип ошибки, обнаруженной компонентом, вызвавшим крах системы.

structured exception handling — структурная обработка исключений ~ разновидность обработки исключений, позволяющая приложениям получать управление при возникновении исключений. При этом приложение может исправить ситуацию, которая привела к исключению, провести раскрутку стека (завершив таким образом выполнение подпрограммы, вызвавшей исключение) или уведомить систему о том, что данное исключение ему не известно, и тогда система продолжит поиск подходящего обработчика для данного исключения.

subsection — подраздел ~ структура с информацией о проецировании каждого раздела файла (атрибуты защиты страниц и т. д.).

subsystem DLL — DLL подсистемы ~ DLL-модуль, транслирующий вызов документированной функции в вызов соответствующего недокументированного системного сервиса Windows.

symbolic link — символьная ссылка ~ механизм косвенной ссылки на имя объекта.

symmetric encryption algorithm — алгоритм симметричного шифрования ~ алгоритм, в котором для шифрования и расшифровки данных используется один и тот же ключ. Такие алгоритмы работают, как правило, очень быстро, и их можно применять для шифрования больших объемов данных, например файлов. Однако слабая сторона алгоритмов симметричного шифрования в том, что защита легко обходится при наличии ключа.

symmetric multiprocessing (SMP) — симметричная многопроцессорная обработка (SMP) ~ многопроцессорная обработка, при которой не выделяется главный процессор, — код операционной системы и пользовательские потоки могут выполняться на любом процессоре. Все процессоры делят единое пространство памяти.

synchronization — синхронизация ~ способность потока синхронизировать свое выполнение, ожидая перехода какого-либо объекта из одного состояния в другое. Поток можно синхронизировать с использованием таких объектов, как процесс, поток, файл, событие, семафор, мьютекс и таймер. Объекты вроде раздела, порта, маркера доступа, каталога объектов, символьной ссылки, профиля и раздела реестра синхронизацию не поддерживают.

synchronous I/O — синхронный ввод-вывод ~ модель ввода-вывода, в которой устройство передает данные и возвращает код состояния только по завершении ввода-вывода. Сразу после этого программа получает доступ к переданным данным. Windows-функции *ReadFile* и *WriteFile* при использовании в простейшей форме выполняются синхронно. Они возвращают управление вызвавшему потоку, только когда полностью завершают операцию ввода-вывода.

system access-control list (SACL) — системный список управления доступом (SACL) ~ список, указывающий, какие операции и каких пользователей следует регистрировать в журнале аудита безопасности.

system audit ACE — ACE системного аудита ~ ACE в SACL, которые наряду с ACE объектов системного аудита определяют, какие операции, выполняемые над объектами конкретными пользователями или группами, подлежат аудиту. ACE объектов системного аудита содержат GUID, указывающий типы объектов или подобъектов, к которым применим данный ACE, и необязательный GUID, контролирующий передачу ACE дочерним объектам конкретных типов. При SACL, равном null, аудит объекта не ведется.

system page table entry (PTE) — системный элемент таблицы страниц (PTE) ~ PTE, используемый при проецировании системных страниц, например пространства ввода-вывода, стеков ядра и списков дескрипторов памяти (MDL).

system service dispatch table — таблица диспетчеризации системных сервисов ~ таблица, в которой каждый элемент содержит указатель на системный сервис, а не на процедуру обработки прерывания.

system services (executive system services) — системные сервисы (сервисы исполнительной системы) ~ встроенные функции операционной системы Windows, которые

можно вызывать из пользовательского режима. Например, *NtCreateProcess* — это внутренний системный сервис, вызываемый Windows-функцией *CreateProcess* при создании нового процесса.

system thread — системный поток ~ поток, выполняемый только в режиме ядра. Системные потоки всегда выполняются в системном процессе (с идентификатором, равным 2). У этих потоков имеются все атрибуты и контексты обычных потоков пользовательского режима (аппаратный контекст, приоритет и др.), но они работают только в режиме ядра в коде, загруженном в системное пространство, — будь то Ntoskrnl.exe или какой-нибудь драйвер устройства. У системных потоков нет пользовательского адресного пространства и поэтому они выделяют память из куч операционной системы вроде пула подкачиваемой или неподкачиваемой памяти.

system worker threads — системные рабочие потоки ~ потоки, создаваемые в процессе System при инициализации системы и существующие только для того, чтобы выполнять работу в интересах других потоков.

system working set — системный рабочий набор ~ физическая память, занятая системным кэшем, пулом подкачиваемой памяти, а также подкачиваемым кодом Ntoskrnl.exe и драйверов устройств. *См. также* process working set.

T

thread — поток ~ некая сущность внутри процесса, получающая процессорное время для выполнения. Поток включает содержимое набора переменных регистров, отражающих состояние процессора, два стека (один используется при выполнении потока в режиме ядра, а другой — при выполнении в пользовательском режиме), закрытую область памяти (используемую подсистемами, библиотеками периода выполнения и DLL), а также уникальный идентификатор потока (внутреннее название — идентификатор клиента).

thread context — контекст потока ~ переменные регистры, стеки и локальная область памяти потока. Поскольку эта информация различна на каждой аппаратной платформе, на которой может работать Windows, соответствующая структура данных (CONTEXT) специфична для конкретной платформы. Фактически CONTEXT, возвращаемая Windows-функцией *GetThreadContext*, является единственной открытой структурой данных в Windows API, зависящей от аппаратной платформы.

transaction — транзакция ~ операция ввода-вывода, изменяющая данные файловой системы или структуру каталогов тома. Отдельные изменения на диске, составляющие транзакцию, выполняются атомарно: в ходе транзакции на диск должны быть внесены все требуемые изменения. Если транзакция прервана сбоем системы, часть изменений, уже внесенных к этому моменту, нужно отменить. Такая отмена называется откатом. После отката база данных возвращается в исходное согласованное состояние, в котором она была до начала транзакции.

transaction table — таблица транзакций ~ таблица, предназначенная для отслеживания начатых, но еще не зафиксированных транзакций. В процессе восстановления результаты подопераций этих транзакций должны быть удалены с диска.

translation look-aside buffer — ассоциативный буфер трансляции ~ кэш центрального процессора, в котором хранятся недавно транслировавшиеся номера виртуальных страниц.

trap — ловушка ~ аппаратный механизм, благодаря которому при прерывании или исключении процессор перехватывает контроль над выполняемым потоком, переключает его из пользовательского режима в режим ядра и передает управление определенной части операционной системы. В Windows процессор передает управление обработчику ловушек.

trap frame — фрейм ловушки ~ структура данных, в которой сохраняется информация о состоянии потока, выполнявшегося на момент прерывания. Эта информация позволяет ядру возобновить выполнение потока после обработки прерывания или исключения. Фрейм ловушки обычно является подмножеством полного контекста потока.

trap handler — обработчик ловушки ~ модуль в ядре, действующий как коммутатор, который перенаправляет исключения и прерывания соответствующему коду.

trusted facility management — управление доверительными отношениями ~ требует поддержки набора ролей (различных типов учетных записей) для разных уровней работы в системе. Например, функции администрирования доступны только по одной группе учетных записей — Administrators (Администраторы).

trusted path functionality — функциональность пути доверительных отношений ~ предотвращает перехват имен и паролей пользователей программами — троянскими конями. Эта функциональность реализована в Windows в виде входной сигнальной комбинации клавиш Ctrl+Alt+Del и не может быть перехвачена непривилегированными приложениями. Такая комбинация клавиш, известная как SAS (secure attention sequence), вызывает диалоговое окно входа в систему, обходя вызов его фальшивого эквивалента из троянского коня.

type object — объект типа ~ внутрисистемный объект, который содержит информацию, общую для каждого экземпляра данного объекта.

U

Unicode ~ международный стандарт кодировки символов, определяющий уникальные 16-битные коды для большинства известных в мире наборов символов.

update record — запись модификации ~ запись, которую NTFS часто помещает в файл журнала. Каждая запись модификации содержит информацию, которая потребуется в случае повторения операции, изменяющей структуру файловой системы.

user mode — пользовательский режим ~ непривилегированный режим работы процессора, в котором выполняются приложения. В этом режиме доступен ограниченный набор интерфейсов; кроме того, ограничен доступ к системным данным. *См. также* kernel mode.

V

VACB index array — массив индексов VACB ~ массив указателей, поддерживаемый диспетчером кэша для отслеживания представлений для данного файла, проецируемых в системный кэш.

view — представление ~ часть объекта «раздел», затребованная процессом. Объект «раздел» может ссылаться на файлы, длина которых намного превышает размер адресного пространства процесса. (Если раздел поддерживается страничным файлом, в нем должно быть достаточно места для размещения всего раздела.) Используя очень большой объект «раздел», процесс может проецировать лишь его представление, которое создается вызовом функции *MapViewOfFile* с указанием проецируемого диапазона. Это позволяет процессам экономить адресное пространство, так как на память проецируются только нужные в данный момент представления объекта «раздел». *См. также* section object.

virtual address control block (VACB) — блок управления виртуальными адресами (VACB) ~ структуры данных, используемые диспетчером кэша для управления системным адресным пространством, на которое проецируются файлы.

virtual address descriptor (VAD) — дескриптор виртуального адреса (VAD) ~ структура данных, поддерживаемая диспетчером памяти для отслеживания виртуальных адресов, зарезервированных в адресном пространстве процесса. Для большей эффективности поиска информации VAD организованы в виде двоичного дерева с автоматической балансировкой (self-balancing binary tree).

virtual cluster number (VCN) — виртуальный номер кластера (VCN) ~ VCN нумеруют кластеры, принадлежащие конкретному файлу (от 0 до *m*). VCN не обязательно должны быть физически непрерывны и могут отображаться на любой LCN в пределах тома.

virtual device drivers (VDD) — драйверы виртуальных устройств ~ драйверы, используемые для эмуляции 16-разрядных программ MS-DOS. Они перехватывают ссылки из программ MS-DOS на порты ввода-вывода и транслируют их в вызовы Windows-функций ввода-вывода. Поскольку Windows — полностью защищенная операционная система, пользовательские программы MS-DOS не имеют прямого доступа к оборудованию и должны обращаться к нему через драйверы устройств реального режима.

virtual memory manager — диспетчер виртуальной памяти ~ реализует виртуальную память — схему управления памятью, в соответствии с которой каждый процесс получает большое закрытое адресное пространство, по своим размерам намного превышающее реальный объем физической памяти.

volume — том ~ один или несколько дисковых разделов, рассматриваемых как единое целое.

volume file — файл тома ~ системный файл (с именем $Volume), содержащий имя тома, версию NTFS, для которой отформатирован том, и бит, устанавливаемый при повреждении диска (он сигнализирует о необходимости запуска утилиты Chkdsk).

volume manager — диспетчер томов ~ обобщенное название драйверов FtDisk и DMIO, поскольку они оба поддерживают одни и те же типы составных томов.

W

wait block — блок ожидания ~ у каждого потока, находящегося в состоянии ожидания, имеется список блоков ожидания — структур данных, представляющих объекты, на которых ждет этот поток. А у каждого объекта диспетчера ядра есть список блоков ожидания — структур данных, представляющих потоки, которые ждут на этом объекте.

WDM drivers — WDM-драйверы ~ драйверы устройств, отвечающие спецификации Windows Driver Model (WDM). WDM требует от драйверов поддержки управления электропитанием, Plug and Play и WMI. WDM реализована в Windows 2000 (и выше), а также в Windows Millennium Edition, поэтому WDM-драйверы этих операционных систем совместимы на уровне исходного кода, а во многих случаях и на уровне двоичного кода. Существует три типа WDM-драйверов: драйверы шин, функциональные драйверы и драйверы фильтров.

Windows API ~ 32-разрядный интерфейс в 32- и 64-разрядных операционных системах семейства Windows.

Windows services — Windows-сервисы ~ механизм запуска процессов, которые предоставляют сервисы и не требуют привязки к интерактивному пользователю. Сервисы аналогичны демонам UNIX и часто реализуются на серверной стороне клиент-серверных приложений.

WindowStation — объект WindowStation ~ объект, содержащий рабочие столы, которые в свою очередь содержат окна. Только один объект WindowStation может быть видим на консоли и принимать пользовательский ввод (с клавиатуры и от мыши). В среде Terminal Services видим один объект WindowStation в каждом сеансе, но все сервисы выполняются как часть консольного сеанса.

Windows drivers — драйверы Windows ~ драйверы устройств, интегрируемые с диспетчерами Plug and Play и электропитания в Windows. К ним относятся драйверы устройств массовой памяти, стеков протоколов и сетевых адаптеров.

Windows Management Instrumentation (WMI) manager — диспетчер WMI ~ компонент исполнительной системы, который позволяет драйверам устройств публиковать информацию о своих рабочих характеристиках и конфигурации и принимать команды от службы WMI пользовательского режима. Потребители WMI-информации могут находиться как на локальной машине, так и на удаленных компьютерах в сети.

work item — рабочий элемент ~ некое задание, помещаемое в специальную очередь (объект диспетчера ядра), проверяемую системными рабочими потоками. Драйвер устройства или компонент исполнительной системы запрашивает сервисы системных рабочих потоков через функцию исполнительной системы *ExQueueWorkItem* или *IoQueueWorkItem*. Рабочий элемент включает указатель на процедуру и параметр, передаваемый потоком этой процедуре при обработке рабочего элемента. Процедура реализуется драйвером устройства или компонентом исполнительной системы, выполняемым при IRQL уровня «passive».

working set — рабочий набор ~ набор виртуальных страниц, резидентных в физической памяти. Существует два типа рабочих наборов — системный и процесса.

working set manager — диспетчер рабочих наборов ~ процедура, выполнямая в контексте системного потока диспетчера настройки баланса; инициирует автоматическое усечение рабочего набора для увеличения объема доступной в системе свободной памяти.

write-back — обратная запись ~ алгоритм кэширования, используемый файловой системой с отложенной записью для повышения производительности. При обратной записи файловая система записывает изменения в кэш, а затем сбрасывает его содержимое на диск (обычно в фоновом режиме).

write-through — сквозная запись ~ алгоритм, применяемый файловой системой FAT для немедленной записи изменений на диск. В отличие от алгоритма точной записи не требует упорядочения операций записи. *См. также* careful write.

Z

zero page thread — поток обнуления страниц ~ системный поток режима ядра (поток 0 в системном процессе), который обнуляет страницы из списка свободных страниц.

Предметный указатель

X

A

П

T

У

Ф

Дэвид Соломон

Дэвид Соломон, президент David Solomon Expert Seminars (*wwwsolsem.com*), в основном занимается разъяснением внутреннего устройства линейки операционных систем Microsoft Windows NT с 1992 года. Он ведет широко известные курсы по внутреннему устройству Windows для тысяч разработчиков и ИТ-специалистов во всем мире. Его клиентами являются все основные компании компьютерной индустрии, в том числе Microsoft. В 1993 году был удостоен награды Microsoft Support Most Valuable Professional (MVP).

Это его четвертая книга. А первой была «Windows NT for OpenVMS Professionals» (Digital Press/Butterworth Heinemann, 1996), в которой программистам и системным администраторам VMS объяснялись принципы работы Windows NT. Прежде чем учредить собственную компанию, Дэвид более девяти лет работал руководителем проекта и был одним из разработчиков операционной системы VMS в Digital Equipment Corporation. Его вторая книга, «Inside Windows NT, Second Edition» (Microsoft Press, 1998), была посвящена внутреннему устройству Windows NT 4.0. Свою третью книгу, «Inside Windows 2000, Third Edition» (Microsoft Press, 2000) он написал в соавторстве с Марком Руссиновичем.

Дэвид регулярно выступает на научно-технических конференциях — Microsoft TechEd и Microsoft PDC; был научным председателем нескольких прошлых конференций по Windows NT. Кроме исследования Windows, Дэвид увлекается парусным спортом, чтением и любит сериал Star Trek.

Марк Руссинович

Марк Руссинович — главный архитектор программного обеспечения и один из учредителей Winternals Software (*www.winternals.com*), компании, специализирующейся на разработке «продвинутого» системного программного обеспечения для Microsoft Windows. Марк является старшим редактором журнала «Windows IT Pro Magazine», где часто пишет колонку «Windows Power Tools». Вместе с Дэвидом Соломоном часто проводит общедоступные и закрытые семинары по внутреннему устройству операционных систем Windows и методикам углубленного анализа проблем для многочисленных компаний и организаций, в том числе Microsoft. Также вместе они подготовили 12-часовой учебный видеокурс по изучению внутреннего устройства Windows, который Microsoft лицензировала у них для внутрикорпоративного использования по всему миру. Марк участвует в работе основных конференций вроде Microsoft Tech Ed, Microsoft IT Forum, Windows IT Pro Magazine's Connections и MCP Magazine's TechMentor.

Марк — обладатель степеней бакалавра (Carnegie Mellon University) и магистра (Rennselaer Polytechnic Institute) в области компьютерных технологий. В 1994 году он получил степень доктора философии в Carnegie Mellon University — также в области компьютерных технологий. До учреждения компании Winternals Software работал в Compuware NuMega Laboratories и в исследовательском центре Thomas J. Watson Research Center корпорации IBM.