Microsoft

МАСТЕР КЛАСС

ВНУТРЕННЕЕ УСТРОЙСТВО Microsoft Windows

6-е издание

М. Руссинович, Д. Соломон

ПИТЕР

Mark Russinovitch, David A. Solomon

Windows Internals

6th Edition

Part 1

Microsoft
PRESS

6-е издание

М. Руссинович, Д. Соломон

ВНУТРЕННЕЕ УСТРОЙСТВО Microsoft Windows

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2013

М. Руссинович, Д. Соломон

Внутреннее устройство Microsoft Windows. 6-е изд.

Серия «Мастер-класс»

Перевел с английского Н. Вильчинский

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Юрченко*
Ведущий редактор	*Ю. Сергиенко*
Литературный редактор	*О. Некруткина*
Художественный редактор	*Л. Адуевская*
Корректор	*В. Листова*
Верстка	*Е. Волошина*

ББК 32.973.2-018.2 УДК 004.451

Руссинович М., Соломон Д.

Р89 Внутреннее устройство Microsoft Windows. 6-е изд. — СПб.: Питер, 2013. — 800 с.: ил. — (Серия «Мастер-класс»).

ISBN 978-5-459-01730-4

Шестое издание этой легендарной книги посвящено внутреннему устройству и алгоритмам работы основных компонентов операционной системы Microsoft Windows 7, а также Windows Server 2008 R2. Определяются ключевые понятия и термины Windows, дается представление об инструментальных средствах, используемых для исследования внутреннего устройства системы, рассматривается общая архитектура и компоненты ОС. Также в книге дается представление о ключевых основополагающих системных и управляющих механизмах Windows, охватываются основные компоненты операционной системы: процессы, потоки и задания; безопасность и работа в сети.

Книга предназначена для системных администраторов, разработчиков сложных приложений и всех, кто хочет понять, как устроена операционная система Windows.

12+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-0735648739 англ.
ISBN 978-5-459-01730-4

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Подписано в печать 04.04.13. Формат 70x100/16. Усл. п. л. 64,500. Тираж 1200. Заказ
Отпечатано в полном соответствии с качеством предоставленных издательством материалов
в Чеховский Печатный Двор. 142300, Чехов, Московская область, г. Чехов, ул. Полиграфистов, д.1.

Краткое оглавление

Оглавление

Введение

Шестое издание книги «Внутреннее устройство Windows» предназначено для подготовленных специалистов в области компьютерной техники (для разработчиков и системных администраторов), желающих разобраться во внутренней работе основных компонентов операционных систем Microsoft Windows 7 и Windows Server 2008 R2. Обладая этими знаниями, разработчики смогут лучше понять обоснование проектных решений при создании приложений, предназначенных для платформы Windows. Эти знания также помогут разработчикам вести отладку сложных проблем. Пользу от этой информации могут также получить и системные администраторы, поскольку понимание принципов работы операционной системы «под капотом» помогает понять вопросы поведения системы при стремлении поднять ее производительность и облегчить решение вопросов диагностики системы при возникновении сбоев в ее работе. Прочитав эту книгу, вы сможете лучше разобраться в работе Windows и в том, каковы причины того или иного ее поведения.

Структура книги

Впервые книга «Внутреннее устройство Windows» была разделена авторами на две части.

Эта книга представляет собой первую часть и начинается двумя главами, в которых определяются ключевые понятия, дается представление об инструментальных средствах, используемых в книге, и рассматривается общая архитектура и компоненты системы. В следующих двух главах дается представление о ключевых основополагающих системных и управляющих механизмах. Также охватываются три основных компонента операционной системы: процессы, потоки и задания; безопасность и работа в сети.

Оригинальное издание второй части было опубликовано издательством Microsoft Press осенью 2012 года и охватывает остальные основные подсистемы: ввод-вывод, хранение данных, управление памятью, диспетчер кэша и файловые системы. Рассмотрены процессы запуска и завершения работы и дано описание анализа аварийного дампа.

Выпуск русского издания второй части данной книги находится в планах издательства «Питер».

История этой книги

Это шестое издание книги, которая сначала называлась «Inside Windows NT» (Microsoft Press, 1992) и была написана Хелен Кастер (Helen Custer) еще до выхода Microsoft Windows NT 3.1. «Inside Windows NT» была первой книгой, написанной о Windows NT и предоставившей проникновение в суть архитектуры и конструкции системы. «Inside Windows NT, Second Edition» (Microsoft Press, 1998) была написана Дэвидом Соломоном (David Solomon). Она дополнила ис-

ходную книгу описанием Windows NT 4.0 и получила существенно возросший уровень технической глубины.

Книга «Inside Windows 2000, Third Edition» (Microsoft Press, 2000) вышла под авторством Дэвида Соломона (David Solomon) и Марка Руссиновича (Mark Russinovich). В нее было добавлено множество новых тем, например запуск и завершение работы, внутренне устройство служб, реестра, драйверов файловой системы и сети. В ней также были рассмотрены изменения, внесенные в ядро Windows 2000, например модель драйверов Windows Driver Model (WDM), Plug and Play, диспетчер энергопотребления, Windows Management Instrumentation (WMI), шифрование, объект задания и службы терминалов. Книга «Windows Internals, Fourth Edition» получила обновления, связанные с выходом Windows XP и Windows Server 2003, и добавления, сконцентрированные на помощи IT-профессионалам в применении их знаний внутреннего устройства Windows, например в использовании основных инструментов из комплекта Windows Sysinternals (www.microsoft.com/technet/sysinternals) и в анализе аварийных дампов. Книга «Windows Internals, Fifth Edition» была обновлена под выход Windows Vista и Windows Server 2008. Новое содержимое включало загрузчик образов, средство отладки в пользовательском режиме и Hyper-V.

Изменения, внесенные в шестое издание

В это самое последнее издание внесены обновления, касающиеся изменений, внесенных в ядро Windows 7 и Windows Server 2008 R2. Были обновлены эксперименты, демонстрирующие изменения, внесенные в инструментарий.

Практические эксперименты

Даже без доступа к исходному коду Windows вы многое можете почерпнуть о внутреннем устройстве Windows из таких инструментальных средств, как отладчик ядра и утилиты из наборов Sysinternals и Winsider Seminars & Solutions. Когда инструментальное средство может быть использовано для показа или демонстрации некоторых аспектов внутреннего поведения Windows, действия, которые можно попытаться выполнить самостоятельно, перечисляются в окнах под названием «ЭКСПЕРИМЕНТ». Они появляются по всей книге, и мы хотим, чтобы вы провели все эти эксперименты по мере чтения книги — наглядное доказательство внутренней работы Windows сильнее отпечатается в вашей памяти, чем простое чтение описания этой работы.

Незатронутые темы

Windows является слишком большой и сложной операционной системой. В этой книге не описывается все подряд, что имеет отношение к внутреннему устройству Windows, но зато делается упор на рассмотрение основных системных компонентов. Например, в этой книге не дается описание COM+, распределенной объектно-ориентированной программной инфраструктуры Windows, или среды Microsoft .NET Framework, которая является основой приложений с управляемым кодом.

Поскольку книга посвящена внутреннему устройству, а не использованию, программированию или системному администрированию, в ней не дается описание того, как нужно использовать, программировать или конфигурировать Windows.

Предупреждение и предостережение

Поскольку в данной книге рассматривается недокументированное поведение внутренней архитектуры и функционирования операционной системы Windows (например, внутренних структур и функций ядра), содержимое может от выпуска к выпуску претерпевать некоторые изменения. (Внешние интерфейсы, например Windows API, не являются предметом несовместимых изменений.)

Под «предметом изменений» не обязательно имеется в виду, что подробности, рассмотренные в данной книге, будут меняться от выпуска к выпуску, но не рассчитывайте на то, что они не претерпят вообще никаких изменений. Любое программное обеспечение, использующее эти недокументированные интерфейсы в будущих выпусках Windows, может оказаться неработоспособным. Хуже того, программы, запускаемые в режиме ядра (например, драйверы устройств) и использующие эти недокументированные интерфейсы, могут при запуске новых выпусков Windows потерпеть системный сбой.

Благодарности

Сначала хочется сказать спасибо Джейми Ханрахан (Jamie Hanrahan) и Брайану Катлину (Brian Catlin) из Azius, LLC за то, что они присоединились к нам в этом проекте — без их помощи книга вряд ли была бы завершена. Они внесли множество обновлений в главы «Безопасность» и «Сеть» и поучаствовали в обновлении глав «Механизмы управления» и «Процессы, потоки и задания». Компания Azius ведет обучение внутреннему устройству Windows и драйверов устройств. Дополнительные сведения можно найти на сайте www.azius.com.

Хочется выразить признательность Алексу Ионеску (Alex Ionescu), который стал полноправным соавтором этого издания. Это реакция на его большой объем работы над пятым изданием и продолжение работы над этим изданием книги.

Спасибо Эрику Трауту (Eric Traut) и Джону ДеВаану (Jon DeVaan) за предоставление Дэвиду Соломону (David Solomon) доступа к исходному коду Windows для его работы над этой книгой, а также для продолжающейся разработки его курсов Windows Internals.

В пятом издании не была выражена благодарность трем ключевым рецензентам за их вклад в это издание: это Арун Кишан (Arun Kishan), Лэнди Ванг (Landy Wang) и Аарон Маргосис (Aaron Margosis) — спасибо им еще раз! И еще раз спасибо Аруну и Лэнди за их подробную рецензию и полезный вклад в это издание.

Эта книга не содержала бы столь глубоких технических подробностей или уровней точности без просмотра, вклада и поддержки со стороны основных специалистов команды разработчиков Microsoft Windows. Поэтому мы хотим поблагодарить следующих людей, предоставивших техническую рецензию и вклад в эту книгу: Greg Cottingham; Joe Hamburg; Jeff Lambert; Pavel Lebedynskiy; Joseph East; Adi Oltean; Alexey Pakhunov; Valerie See.

За главу «Сети» хочется выразить особую благодарность Джанлуиджи Нуска (Gianluigi Nusca) и Тому Джолли (Tom Jolly), кто действительно сделал больше, чем того требовал их служебный долг: Джанлуиджи за его неоценимую помощь по материалу о BranchCache и массу предложений (и множество абзацев написанного им материала) и Тому Джолли не только за его собственную рецензию и предложения (которые были превосходными), но и за привлечение к составлению рецензии многих других разработчиков. Вот все, кто участвовал в написании рецензии и сделал свой вклад в главу «Сеть»: Roopesh Battepati; Molly Brown; Greg Cottingham; Dotan Elharrar; Eric Hanson; Tom Jolly; Manoj Kadam; Greg Kramer; David Kruse; Jeff Lambert; Darene Lewis; Dan Lovinger; Gianluigi Nusca; Amos Ortal; Ivan Pashov; Ganesh Prasad; Paul Swan; Shiva Kumar Thangapandi.

Амос Ортал (Amos Ortal) и Дотан Элхаррар (Dotan Elharrar) оказали особую помощь в описании NAP, а со стороны Шива Кумар Сангапанди (Shiva Kumar Thangapandi) была предоставлена существенная помощь в описании EAP.

Детальная проверка, проведенная Кристофом Назарре (Christophe Nasarre), общим техническим рецензентом, стала существенным вкладом в техническую точность и последовательность изложения материала этой книги.

Хочется еще раз поблагодарить Ильфака Гуилфанова (Ilfak Guilfanov) из компании Hex-Rays (www.hex-rays.com) за лицензии IDA Pro Advanced и Hex-Rays, которые были выделены Алексу Ионеску (Alex Ionescu), благодаря чему он смог ускорить свой анализ ядра Windows.

И наконец, авторы хотят поблагодарить замечательный коллектив Microsoft Press, воплотивший эту книгу в реальность.

И последнее по очередности, но не по значимости спасибо следует сказать Бену Райану (Ben Ryan), издателю Microsoft Press, который не утратил веры в важность предоставления подобного уровня детализации, касающейся операционной системы Windows своим читателям!

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на веб-сайте издательства http://www.piter.com.

Глава 1. Общие представления и инструментальные средства

В этой главе будут даны ключевые понятия и термины операционной системы Microsoft Windows, которые будут использоваться по всей книге. Это Windows API, процессы, потоки, виртуальная память, режим ядра и пользовательский режим, объекты, дескрипторы, система безопасности и реестр. Будут также показаны инструментальные средства, которые могут использоваться для исследования таких внутренних компонентов Windows, как отладчик ядра, монитор производительности и ключевой инструментарий из состава Windows Sysinternals (www.microsoft.com/technet/sysinternals). Кроме этого вы узнаете, как пользоваться наборами инструментов для создания драйверов — Windows Driver Kit (WDK) — и для разработки программного обеспечения — Windows Software Development Kit (SDK) — в качестве ресурсов для поиска дополнительной информации о внутренних компонентах Windows.

Материал этой главы нужно обязательно усвоить, потому что это необходимо для понимания остальных глав книги.

Версии операционной системы Windows

В этой книге рассматриваются самые последние клиентские и серверные версии операционных систем Microsoft Windows: Windows 7 (32- и 64-разрядные версии) и Windows Server 2008 R2 (только 64-разрядная версия). Если нет специальных оговорок, предоставленная информация относится ко всем версиям. В табл. 1.1 перечислены названия продуктов Windows, их внутренние номера версий и даты выпусков.

Таблица 1.1. Выпуски операционной системы Windows

Название продукта	Внутренний 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 Vista	6.0 (сборка 6000)	Январь 2007 г.
Windows Server 2008	6.0 (сборка 6001)	Март 2008 г.
Windows 7	6.1 (сборка 7600)	Октябрь 2009 г.
Windows Server 2008 R2	6.1 (сборка 7600)	Октябрь 2009 г.

ПРИМЕЧАНИЕ

Цифра «7» в имени продукта «Windows 7» не относится к внутреннему номеру версии, она просто является признаком поколения. В действительности, чтобы свести к минимуму проблемы совместимости приложений, для Windows 7 настоящим номером версии считается 6.1, что и показано в табл. 1.1. Это дает возможность тем приложениям, которые проводят проверку на основной номер версии, продолжать вести себя в Windows 7 точно так же, как это делалось в Windows Vista. Фактически, и у Windows 7, и у Server 2008 R2 одинаковые номера версий и сборок, потому что они были созданы из одной и той же кодовой основы Windows.

Основные термины и понятия

В этой книге будут встречаться ссылки на некоторые структуры и понятия, которые могут быть неизвестны отдельным читателям. В этом разделе мы дадим определения тем терминам, которые будут встречаться по всей книге. Ознакомьтесь с ними прежде, чем перейти к следующим главам.

Windows API

Интерфейс прикладного программирования Windows API (application programming interface) является интерфейсом системного программирования в пользовательском режиме для семейства операционных систем Windows. До выхода 64-разрядных версий Windows программный интерфейс для 32-разрядных версий операционных систем Windows назывался Win32 API, чтобы его можно было отличить от исходной 16-разрядной версии Windows API (которая служила интерфейсом программирования для начальных 16-разрядных версий Windows). В данной книге термин *Windows API* будет относиться как к 32-, так и к 64-разрядным интерфейсам программирования в среде Windows.

ПРИМЕЧАНИЕ

Описание Windows API можно найти в документации по набору инструментальных средств разработки программного обеспечения — Windows Software Development Kit (SDK). (См. далее в этой главе раздел «Windows Software Development Kit».) Эта документация доступна на веб-сайте www.msdn.microsoft.com. Она также включена со всеми уровнями подписки в сеть Microsoft Developer Network (MSDN), предназначенную для разработчиков. Дополнительную информацию можно найти по адресу www.msdn.microsoft.com. Великолепное описание процесса программирования с помощью базового Windows API дано в книге «Windows via C/C++», пятое издание, написанной Джеффри Рихтером (Jeffrey Richter) и Кристофером Назарре (Christophe Nasarre) (Microsoft Press, 2007).

Windows API состоит из нескольких тысяч вызываемых функций, которые разбиты на следующие основные категории:

- Базовые службы (Base Services).
- Службы компонентов (Component Services).
- Службы пользовательского интерфейса (User Interface Services).

- Графические и мультимедийные службы (Graphics and Multimedia Services).
- Обмен сообщениями и совместная работа (Messaging and Collaboration).
- Сеть (Networking).
- Веб-службы (Web Services).

В этой книге основное внимание уделяется внутренним составляющим ключевых базовых служб: процессам и потокам, управлению памятью, вводу-выводу и безопасности.

А ЧТО МОЖНО СКАЗАТЬ НАСЧЕТ ТЕХНОЛОГИИ .NET?

Microsoft .NET Framework состоит из библиотеки классов под названием Framework Class Library (FCL) и управляемой среды выполнения кода — Common Language Runtime (CLR). CLR обладает функциями своевременной компиляции, проверки типов, сборки мусора и обеспечения безопасности доступа к коду. Предлагая эти функции, CLR предоставляет среду разработки, повышающую производительность работы программистов и сокращающую количество наиболее распространенных ошибок программирования. Отличное описание .NET Framework и архитектуры ядра этой платформы можно найти в книге «CLR via C#», третье издание, написанной Джеффри Рихтером (Jeffrey Richter) (Microsoft Press, 2010).

Среда CLR реализована, как классический COM-сервер, код которого находится в стандартной Windows DLL-библиотеке, предназначенной для работы в пользовательском режиме. Фактически все компоненты .NET Framework реализованы как стандартные Windows DLL-библиотеки пользовательского режима, наложенные поверх неуправляемых функций Windows API. (Ни один из компонентов .NET Framework не работает в режиме ядра.) Взаимоотношения между этими компонентами показаны на рис. 1.1.

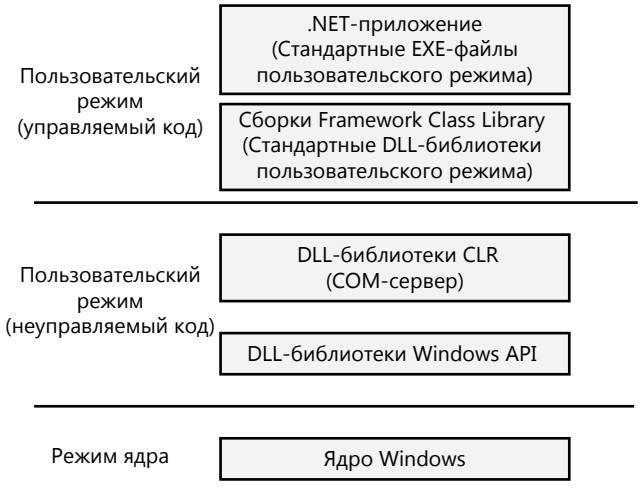


Рис. 1.1. Взаимоотношения между компонентами .NET Framework

Службы, функции и стандартные программы

Некоторые термины в пользовательской и программной документации Windows в разных контекстах имеют разные значения. Например, слово *служба* может

относиться к вызываемой в операционной системе стандартной подпрограмме, драйверу устройства или к обслуживающему процессу. Что именно означают те или иные термины в этой книге, показано в следующем списке:

- **Функции Windows API.** Документированные, вызываемые подпрограммы в Windows API. Например, CreateProcess, CreateFile и GetMessage.
- **Собственные системные службы** (или системные вызовы). Недокументированные базовые службы в операционной системе, вызываемые при работе в пользовательском режиме. Например, NtCreateUserProcess является внутренней службой, которую функция Windows CreateProcess вызывает для создания нового процесса. Определение системного вызова дано в разделе «Диспетчеризация системных служб» (см. главу 3).
- **Функции поддержки ядра** (или подпрограммы). Подпрограммы внутри операционной системы Windows, которые могут быть вызваны только из режима ядра (определение которому будет дано далее в этой главе). Например, ExAllocatePoolWithTag является подпрограммой, вызываемой драйверами устройств для выделения памяти из системных динамически распределяемых областей Windows (называемых пулами).
- **Службы Windows.** Процессы, запускаемые Диспетчером управления службами (Windows service control manager). Например, служба Диспетчер задач запускается в виде процесса, работающего в пользовательском режиме, в котором поддерживается команда at (аналогичная UNIX-командам at или cron). (Примечание: хотя в реестре драйверы устройств Windows определены как «службы», в этой книге в таком качестве они не упоминаются.)
- **Библиотеки DLL** (dynamic-link libraries — динамически подключаемые библиотеки). Набор вызываемых подпрограмм, связанных вместе в виде двоичного файла, который может быть загружен в динамическом режиме приложениями, которые используют эти подпрограммы. В качестве примера можно привести Msvcrt.dll (библиотеку времени выполнения для приложений, написанных на языке C) и Kernel32.dll (одну из библиотек подсистемы Windows API). DLL-библиотеки широко используются компонентами и приложениями Windows, которые работают в пользовательском режиме. Преимущество, предоставляемое DLL-библиотеками по сравнению со статическими библиотеками, заключается в том, что они могут использоваться сразу несколькими приложениями, и Windows обеспечивает наличие в памяти только одной копии кода DLL-библиотеки для тех приложений, в которых имеются ссылки на эту библиотеку. Следует заметить, что неисполняемые .NET-сборки компилируются как DLL-библиотеки, но без каких-либо экспортируемых подпрограмм. CLR анализирует скомпилированные метаданные для доступа к соответствующим типам и элементам классов.

ИСТОРИЯ WIN32 API

Интересно, что Win32 не планировался в качестве исходного интерфейса программирования для той системы, которая в ту пору называласьWindows NT. Поскольку проект Windows NT запускался в качестве замены для OS/2 версии 2, первоначальным интерфейсом программирования был 32-разрядный OS/2 Presentation Manager API. Но через год после запуска проекта произошел взлет поступившей в продажу Microsoft Windows 3.0. В резуль-

тате этого Microsoft сменила направление и сделала Windows NT будущей заменой семейства продуктов Windows, а не заменой OS/2. В связи с этим и возникла необходимость выработать спецификацию Windows API — до этого, в Windows 3.0, API существовал только в виде 16-разрядного интерфейса.

Хотя в Windows API намечалось введение множества новых функций, недоступных в Windows 3.1, Microsoft решила сделать новый API, по возможности, максимально совместимым по именам, семантике и используемым типам данных с 16-разрядным Windows API, чтобы облегчить бремя переноса существующих 16-разрядных Windows-приложений в Windows NT. Этим, собственно, и объясняется тот факт, что многие имена функций и интерфейсов могут показаться непоследовательными: так нужно было для обеспечения совместимости нового Windows API со старым 16-разрядным Windows API.

Процессы, потоки и задания

Хотя при поверхностном взгляде программы и процессы похожи друг на друга, на самом деле они в корне различаются. Программа — это статическая последовательность инструкций, в то время как процесс — это контейнер для набора ресурсов, используемых при выполнении экземпляра программы. На самом высоком уровне абстракции Windows-процесс включает в себя следующее:

- закрытое виртуальное адресное пространство, являющееся набором адресов виртуальной памяти, которым процесс может воспользоваться;
- исполняемую программу, определяющую исходный код и данные, и отображаемую на виртуальное адресное пространство процесса;
- перечень открытых дескрипторов (описателей) различных системных ресурсов — семафоров, коммуникационных портов и файлов, доступных всем потокам процесса;
- связанную с процессом среду безопасности, называемую маркером доступа, идентифицирующим пользователя, группы безопасности, права доступа, виртуализированное состояние системы управления учетными записями пользователей — User Account Control (UAC), сессию и ограниченное состояние учетной записи пользователя;
- уникальный идентификатор, называемый идентификатором процесса, — process ID (внутренняя часть идентификатора называется идентификатором клиента — client ID);
- как минимум один поток выполнения (хотя возможен и абсолютно бесполезный «пустой» процесс).

Каждый процесс также указывает на свой родительский процесс или процесс-создатель. Если родительского процесса больше нет, эта информация не обновляется. Поэтому процесс может указывать на уже несуществующий родительский процесс. Это не создает никаких проблем, поскольку от актуальности этой информации ничего не зависит. При использовании программы исследования процессов ProcessExplorer берется в расчет время запуска родительского процесса, чтобы избежать присоединения дочернего процесса на основе повторно используемого идентификатора процесса. Такое поведение иллюстрируется экспериментом, описанным ниже.

ЭКСПЕРИМЕНТ: ПРОСМОТР ДЕРЕВА ПРОЦЕССОВ

Одним из уникальных атрибутов, относящихся к процессу и неотображаемых большинством инструментальных средств, является идентификатор родительского процесса или процесса-создателя (parent or creator process ID). Это значение можно извлечь с помощью Системного монитора (Performance Monitor) или программным способом, путем запроса Creating Process ID. Дерево процессов может показать такое средство, как Tlist.exe (из состава предназначенных для Windows средств отладки Debugging Tools), используемое с ключом /t. Рассмотрим пример вывода, полученного в результате выполнения команды tlist /t:

```
C:\>tlist /t
System Process (0)
System (4)
  smss.exe (224)
csrss.exe (384)
csrss.exe (444)
  conhost.exe (3076) OleMainThreadWndName
winlogon.exe (496)
wininit.exe (504)
  services.exe (580)
    svchost.exe (696)
    svchost.exe (796)
    svchost.exe (912)
    svchost.exe (948)
    svchost.exe (988)
    svchost.exe (244)
      WUDFHost.exe (1008)
      dwm.exe (2912) DWM Notification Window
    btwdins.exe (268)
    svchost.exe (1104)
    svchost.exe (1192)
    svchost.exe (1368)
    svchost.exe (1400)
    spoolsv.exe (1560)
    svchost.exe (1860)
    svchost.exe (1936)
    svchost.exe (1124)
    svchost.exe (1440)
    svchost.exe (2276)
    taskhost.exe (2816) Task Host Window
    svchost.exe (892)
  lsass.exe (588)
  lsm.exe (596)
explorer.exe (2968) Program Manager
  cmd.exe (1832) Administrator: C:\Windows\system32\cmd.exe - "c:\tlist.exe"  /t
    tlist.exe (2448)
```

Чтобы показать взаимоотношения каждого процесса с его родительскими и дочерними процессами, применяются отступы. Процессы, родители

которых прекратили свое существование, выровнены по левому краю (как Explorer.exe в предыдущем примере), поскольку, даже при наличии прародительского процесса, способов обнаружения связи с ним просто не существует. Windows сохраняет только идентификатор процесса-создателя и не дает ссылок на создателя этого создателя и т. д.
Чтобы продемонстрировать тот факт, что Windows не отслеживает более одного идентификатора родительского процесса, выполните следующие действия:

1. Откройте окно командной строки.
2. Наберите title Parent, чтобы изменить заголовок окна на «Parent» (родительский).
3. Наберите start cmd (что приведет к запуску второго окна командной строки).
4. Наберите во втором окне командной строки title Child, чтобы изменить заголовок окна на «Child» (дочерний).
5. Откройте Диспетчер задач.
6. Наберите во втором окне командной строки mspaint (команду, запускающую Microsoft Paint).
7. Снова обратитесь ко второму окну командной строки и наберите exit. (Заметьте, что Paint остается в рабочем состоянии.)
8. Перейдите в Диспетчер задач.
9. Щелкните на вкладке Приложения.
10. Щелкните правой кнопкой мыши на задаче Parent и выберите пункт Перейти к процессу.
11. Щелкните правой кнопкой мыши на процессе cmd.exe и выберите пункт Завершить дерево процессов.
12. В окне подтверждения Диспетчера задач щелкните на кнопке Завершить дерево процессов.

Первое окно командной строки исчезнет, но по-прежнему можно будет наблюдать окно программы Paint, поскольку оно было потомком во втором поколении завершенного процесса командной строки. Поскольку промежуточный процесс (родительский по отношению к Paint) был завершен, связь между родительским процессом и его потомком во втором поколении была утрачена. ■

Для просмотра процессов и информации о них (а также для внесения изменений) используется несколько инструментальных средств. В описанных ниже экспериментах иллюстрируются различные виды информации о процессах, которые могут быть получены с помощью некоторых из этих средств. Большинство этих средств входит в состав самой Windows, а также в состав отладочного комплекта Debugging Tools for Windows и в состав Windows SDK, но есть и другие автономные средства под маркой Sysinternals. Многие из этих средств показывают частично совпадающие поднаборы сведений об основных процессах и потоках, которые иногда идентифицируются по-разному.

Наверное, самым востребованным средством изучения активности процессов является Диспетчер задач[1]. В ходе следующего эксперимента будет показана

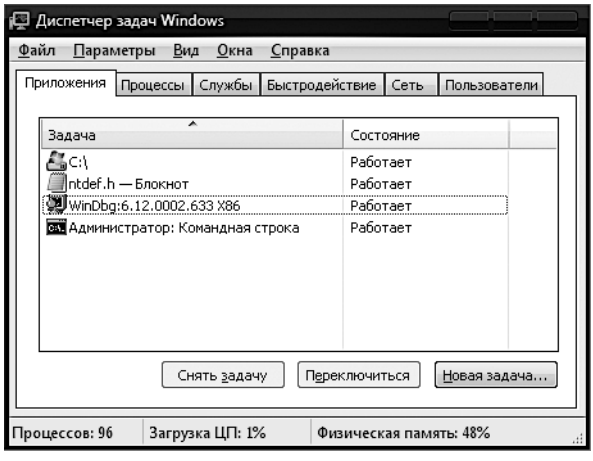
[1] Поскольку в отношении ядра Windows такое понятие, как «задача», не используется, название этого средства — Диспетчер задач — кажется немного странным.

разница между тем, что в списках Диспетчера задач называется приложениями, и процессами.

ЭКСПЕРИМЕНТ: ПРОСМОТР ИНФОРМАЦИИ О ПРОЦЕССАХ С ПОМОЩЬЮ ДИСПЕТЧЕРА ЗАДАЧ

Встроенный в Windows Диспетчер задач предоставляет краткий список процессов, идущих в системе. Запустить этот диспетчер можно одним из четырех способов: 1) нажатием клавиш Ctrl+Shift+Esc, 2) щелчком правой кнопки мыши на панели задач с последующим выбором пункта Запустить диспетчер задач, 3) нажатием клавиш Ctrl+Alt+Delete с последующим щелчком на кнопке Запустить диспетчер задач, или 4) запуском исполняемой программы Taskmgr.exe. Чтобы увидеть перечень процессов, нужно после запуска Диспетчера задач щелкнуть на вкладке Процессы. Обратите внимание на то, что процессы идентифицируются по именам тех образов, экземплярами которых они являются. В отличие от некоторых объектов Windows, процессам нельзя давать глобальные имена. Чтобы посмотреть дополнительную информацию, выберите в меню Вид пункт Показать столбцы и отметьте те столбцы, которые нужно добавить.

Вполне очевидно, что на вкладке Процессы Диспетчера задач показывается список процессов, а вот о содержимом вкладки Приложения с такой же долей очевидности судить нельзя. Там показывается список видимых окон верхнего уровня на всех Рабочих столах интерактивного оконного терминала, к которому вы подключены. (По умолчанию есть только один интерактивный Рабочий стол, но приложение может создать и больше, если воспользуется функцией Windows CreateDesktop, как это сделано в средстве Sysinternals Desktops.) В столбце Состояние показывается, находится ли поток, являющийся владельцем окна, в состоянии ожидания сообщения от этого окна. Состояние «Работает» означает, что поток ожидает поступления в это окно какого-нибудь ввода, а состояние «Не отвечает» означает, что поток не ждет поступления ввода в окно (например, поток может быть запущен или же находиться в ожидании ввода-вывода или в ожидании изменения состояния какого-нибудь объекта синхронизации Windows).



На вкладке Приложения можно сопоставить задачу тому процессу, который является владельцем потока, являющегося владельцем окна задачи. Для этого нужно щелкнуть правой кнопкой мыши на имени задачи и выбрать пункт Перейти к процессу, как показано в предыдущем эксперименте со средством tlist. ■

Инструментальное средство Process Explorer из арсенала Sysinternals предоставляет больше подробностей о процессах и потоках, чем другие доступные средства, поэтому вы увидите его в ряде экспериментов, рассматриваемых в данной книге. Можно упомянуть следующие, допускаемые Process Explorer уникальные возможности показа той или иной информации или выполнения действий:

- демонстрация маркера доступа к процессу (в виде перечня групп, прав доступа и состояния виртуализации);
- выделение изменений в списке процессов и потоков;
- вывод списка служб внутри процессов, являющихся их хозяевами, включая демонстрацию имен и описаний этих служб;
- показ процессов, являющихся частью задания и подробностей задания;
- показ процессов, являющихся хозяевами .NET-приложений и специфичных для .NET-технологии подробностей (таких как список экземпляров класса AppDomain, загруженных сборок и счетчиков производительности CLR);
- показ времени запуска процессов и потоков;
- показ полного списка отображенных на память файлов (не только DLL-библиотек);
- приостановка процесса или потока;
- завершение отдельного потока;
- простота определения тех процессов, которые потребляли за определенный период времени основную часть времени центрального процессора. (Системный монитор (Performance Monitor) может показывать процесс использования центрального процессора для заданного набора процессов, но он не будет автоматически показывать процессы, созданные после запуска сеанса мониторинга производительности, это может сделать только ручная трассировка в двоичном формате вывода.)

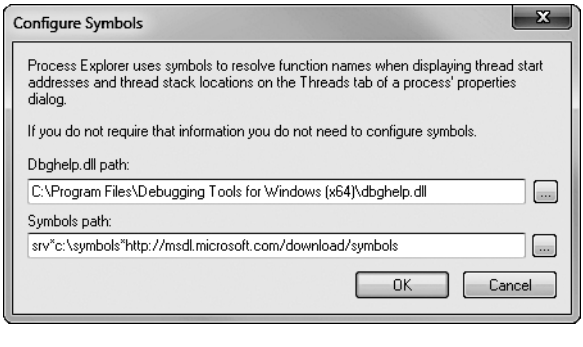
Process Explorer также обеспечивает легкий доступ к информации, собранной в одном месте. Вы можете просмотреть:

- дерево процессов (с возможностью свертывать части дерева);
- открытые в процессе дескрипторы (включая безымянные дескрипторы);
- список имеющихся в процессе DLL-библиотек (и отображенных на память файлов);
- активность потоков в процессе;
- стеки потоков пользовательского режима и режима ядра (включая отображение адресов на имена с использованием библиотеки Dbghelp.dll, поставляемой с отладочным пакетом Debugging Tools для Windows);
- более точный процентный показатель использования центрального процессора, вычисляемый с помощью счетчика циклов потока (как объясняется в главе 5 «Процессы, потоки и задания», по сравнению с обычным представлением, это еще более точное представление об активности центрального процессора);
- уровень целостности;
- подробности управления памятью, такие как зафиксированная пиковая нагрузка (peak commit charge), страницы памяти ядра (kernel memory paged) и лимиты резидентного пула (nonpaged pool limits) — другие инструментальные средства показывают только текущий размер.

А теперь проведем начальный эксперимент с использованием средства Process Explorer.

ЭКСПЕРИМЕНТ: ПРОСМОТР ПОДРОБНОСТЕЙ ПРОЦЕССА С ПОМОЩЬЮ PROCESS EXPLORER

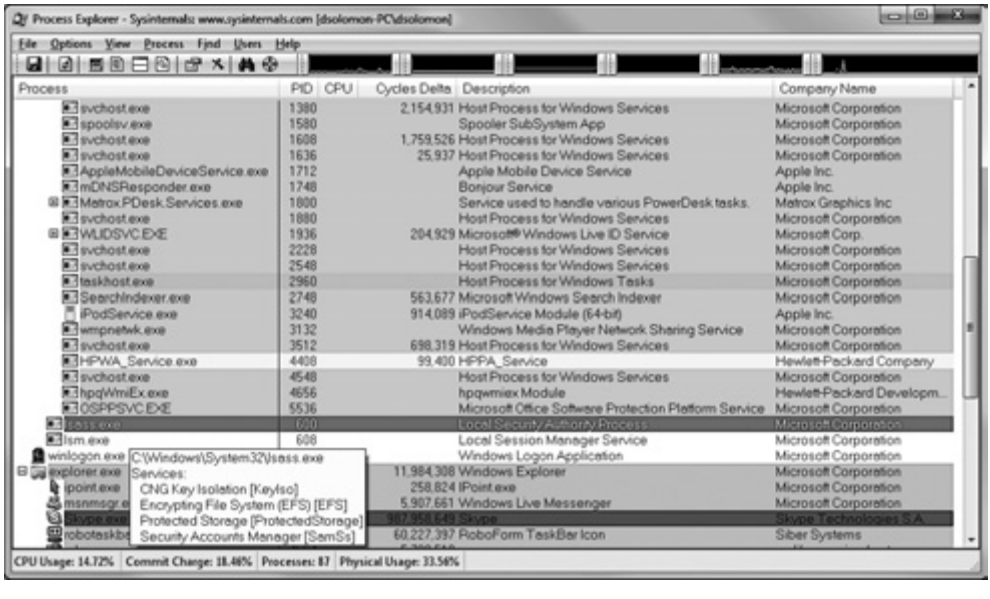
Загрузите самую последнюю версию Process Explorer с сайта Sysinternals и запустите ее. При первом запуске вы получите сообщение о неправильной настройке символов. При правильной настройке Process Explorer может получить доступ к информации о символах для вывода символического имени стартовой функции потока и функций в стеке вызовов потока (доступном после двойного щелчка на имени процесса и щелчка на вкладке Threads (Потоки)). Это поможет определить, что делают потоки внутри процесса. Для доступа к символам у вас должен быть установлен пакет средств для отладки Debugging Tools для Windows (его описание будет дано чуть позже). После его установки щелкните на пункте Options (Дополнительные параметры), выберите пункт Configure Symbols (Настройка символов) и заполните поле пути к библиотеке Dbghelp.dll в папке Debugging Tools, а также укажите правильный путь к символам. Например, для 64-разрядной системы вполне подойдет следующая настройка.



В предыдущем примере для доступа к символам и копирования файлов символов с целью хранения этих файлов на локальной машине в каталоге c:\symbols используется сервер востребованных символов. Более подробная информация по настройкам использования сервера символов дана на сайте http://msdn.microsoft.com/en-us/windows/hardware/gg462988.aspx.

При запуске средства Process Explorer оно по умолчанию показывает дерево процессов. У него также имеется дополнительная нижняя панель, в которой могут быть показаны открытые дескрипторы или отображенные на память DLL-библиотеки и другие, отображенные на память файлы. (Все это исследуется в главе 3 «Системные механизмы».) Также это средство дает подсказки для некоторых видов хост-процессов (hosting processes):

- При перемещении указателя мыши над именем хост-процесса служб (Svchost.exe) показываются службы внутри этого процесса.
- Показываются задачи COM-объектов внутри процесса Taskeng.exe (запускаемого Диспетчером задач (Task Scheduler)).
- Показывается целевой объект процесса Rundll32.exe (используется для таких объектов, как элементы Панели управления (Control Panel)).
- Показывается COM-объект, размещающийся внутри процесса Dllhost.exe.
- Показываются процессы вкладок Internet Explorer.
- Показываются хост-процессы консоли.



Оценить некоторые основные возможности Process Explorer вам помогут следующие действия:

1. Обратите внимание на то, что хост-процессы служб по умолчанию выделены розовым фоном. Ваши собственные процессы выделены синим фоном. (Цвета можно настроить по-другому.)
2. Проведите указателем мыши над отображениями имен процессов и обратите внимание на то, что в окне подсказки показывается полное пу-

тевое имя. Как уже говорилось, для процессов конкретного типа в подсказке выводятся и дополнительные сведения.

3. Щелкните на пунктах View (Вид), Select Columns (Выбрать столбцы) и во вкладке Process Image (Отображение процесса) установите флажок Image Path (Отображение пути).
4. Отсортируйте информацию, щелкнув на заголовке столбца Process (Процесс), и обратите внимание на то, что отображение дерева исчезло. (Доступен либо просмотр дерева, либо просмотр информации, отсортированной по любому из показанных столбцов.) Щелкните еще раз для сортировки в порядке следования имен от Z до A. Затем щелкните еще раз, чтобы вернуться к просмотру дерева процессов.
5. Чтобы просматривать только свои процессы, щелкните на пункте View (Вид) и снимите флажок Show Processes From All Users (Показывать процессы от всех пользователей).
6. Щелкните на пунктах Options (Дополнительные параметры), Difference Highlight Duration (Продолжительность различных подсветок) и измените значение на 5 секунд. Затем запустите новый (любой) процесс и обратите внимание на то, что новый процесс будет выделен зеленым фоном в течение 5 секунд. Завершите этот новый процесс и обратите внимание, что он остается выделенным красным фоном в течение 5 секунд, пока не исчезнет из перечня. Этот эффект помогает заметить в вашей системе созданные и завершенные процессы.

И наконец, щелкните дважды на имени процесса и исследуйте различные вкладки, доступные в окне свойств процесса. (Ссылки на эти вкладки будут встречаться в различных экспериментах, предлагаемых в данной книге, там же будут даны объяснения, касающиеся выводимой в них информации.) ■

Поток, выполнение которого планируется операционной системой Windows, является составляющей того или иного процесса. Без него программа, запустившая процесс, не сможет работать. Поток включает в себя следующие основные компоненты:

- Содержимое набора регистров центрального процессора, отображающее его состояние.
- Два стека — один, используемый потоком при выполнении кода в режиме ядра, и один, используемый при выполнении кода в пользовательском режиме.
- Закрытую область хранения, называемую локальным хранилищем потока — thread-local storage (TLS) для использования подсистемами, библиотеками времени выполнения и DLL-библиотеками.
- Уникальный идентификатор, называемый идентификатором потока — *thread ID* (часть внутренней структуры, называемая идентификатором клиента — *client ID* — идентификаторы процессов и идентификаторы потоков образуются из одного и того же пространства имен, поэтому они никогда не перекрываются).
- Иногда у потоков имеется свой собственный контекст безопасности, или маркер (token), часто используемый многопоточными серверными приложениями, который является олицетворением контекста безопасности обслуживаемых этими приложениями клиентов.

Подвергаемые изменениям регистры, стеки и закрытая область хранения называются контекстом потока. Поскольку эта информация для каждой машинной

архитектуры, в среде которой запущена Windows, бывает разной, структура, если это необходимо, также зависит от той или иной архитектуры. Доступ к этой, специфической для конкретной архитектуры, информации (называемой блоком CONTEXT) предоставляется Windows-функцией GetThreadContext.

ПРИМЕЧАНИЕ

В потоках 32-разрядных приложений, запущенных под управлением 64-разрядной версии Windows, будут содержаться блоки как 32-разрядного, так и 64-разрядного контекста, которые Wow64 будет использовать при необходимости для переключения приложения из работы в 32-разрядном режиме в работу в 64-разрядном режиме. У этих потоков будут два стека для работы в пользовательском режиме и два CONTEXT-блока, а обычные функции Windows API будут возвращать 64-разрядный контекст. Но функция Wow64GetThreadContext будет возвращать 32-разрядный context. Дополнительная информация о Wow64 дана в главе 3.

Волокна и потоки планировщика пользовательского режима

Поскольку при переключении выполнения с одного потока на другой задействуется планировщик ядра, такая операция может обойтись весьма дорого, особенно если два потока часто переключаются между собой. В Windows реализованы два механизма сокращения «накладных расходов»: волокна и использование планировщика пользовательского режима — user-mode scheduling (UMS).

Волокна позволяют приложению осуществлять планирование работы своих собственных «потоков», не полагаясь на встроенный в Windows механизм планирования на основе приоритетов. Волокна часто называют «облегченными» потоками, и, с точки зрения планирования работы, ядру они не видны, поскольку реализуются в пользовательском режиме с помощью библиотеки Kernel32.dll. Чтобы воспользоваться волокнами, сначала нужно вызвать Windows-функцию преобразования потока в волокно — ConvertThreadToFiber. Эта функция преобразует поток в запущенное волокно. Впоследствии только что созданное волокно может с помощью функции CreateFiber создавать дополнительные волокна. (У каждого волокна может быть свой собственный набор волокон.) Но в отличие от потока волокно не приступает к выполнению своего кода, пока оно не будет выбрано с помощью функции SwitchToFiber. Новое волокно выполняется до тех пор, пока оно существует, или до тех пор, пока в нем не будет вызвана функция SwitchToFiber, которая выберет для выполнения другое волокно. Дополнительные сведения можно найти в документации Windows SDK по функциям волокон.

UMS-потоки, доступные только на 64-разрядных версиях Windows, имеют те же основные преимущества, что и волокна, но избавлены от многих недостатков, присущих волокнам. UMS-потоки имеют свое собственное состояние потоков ядра, и поэтому они видимы ядру, которое позволяет нескольким UMS-потокам выдавать блокирующие системные вызовы, совместно использовать и вести борьбу за ресурсы и иметь состояние для каждого потока. Но когда двум и более UMS-потокам требуется работать только в пользовательском режиме, они могут периодически переключать контексты выполнения (за счет уступок одного потока другому) без участия планировщика: переключение контекста происходит

в пользовательском режиме. С точки зрения ядра ничего не меняется и продолжается выполнение все того же потока. Когда UMS-поток выполняет операцию, требующую входа в ядро (например, системный вызов), он переключается на выделенный ему поток режима ядра (это называется непосредственным переключением контекста — directed context switch). Дополнительная информация по UMS дана в главе 5.

Хотя у потоков имеется свой собственный контекст выполнения, каждый поток внутри какого-нибудь процесса использует общее виртуальное адресное пространство этого процесса (вдобавок ко всем остальным ресурсам, принадлежащим процессу). Таким образом, все потоки в процессе имеют полноправный доступ к виртуальному адресному пространству процесса. Но потоки не могут случайно сослаться на адресное пространство другого процесса, пока этот другой процесс не сделает часть своего закрытого адресного пространства общим разделом памяти[1], или пока у одного процесса не будет прав на открытие другого процесса для использования таких функций памяти, касающихся обоих процессов, как ReadProcessMemory и WriteProcessMemory.

Как показано на рис. 1.2, кроме закрытого адресного пространства и одного или нескольких потоков, у каждого процесса есть контекст безопасности и список открытых дескрипторов таких объектов ядра, как файлы, общие разделы памяти или один из объектов синхронизации из разряда мьютексов, событий или семафоров.

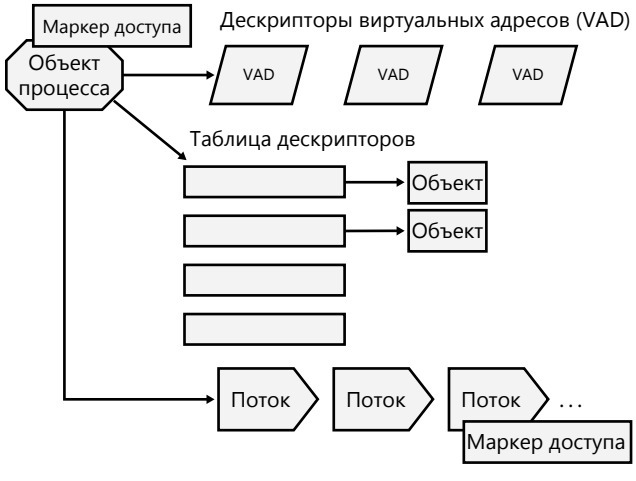


Рис. 1.2. Процесс и его ресурсы

Контекст безопасности каждого процесса хранится в объекте, который называется *маркером доступа* (access token). Маркер доступа процесса содержит идентификацию безопасности и полномочия процесса. По умолчанию потоки не имеют своего собственного маркера доступа, но они могут получить такой маркер, позволяющий отдельным потокам имитировать контекст безопасности другого процесса, включая процессы на удаленной системе Windows, не оказывая

[1] В Windows API этот раздел памяти называется объектом, проецируемым на файл.

при этом никакого влияния на другие потоки процесса. (Более подробно вопросы безопасности процессов и потоков рассмотрены в главе 6 «Безопасность».)

Дескрипторы виртуального адресного пространства — virtual address descriptors (VAD) являются структурами данных, используемых диспетчером памяти для отслеживания виртуальных адресов, используемых процессом.

Windows предоставляет расширение модели процесса, называемое *заданием* (job). Основная функция объектов заданий заключается в том, чтобы управлять группами процессов как единым целым и осуществлять на них одновременное воздействие. Объект задания позволяет управлять конкретными атрибутами и предоставляет ограничения для процесса или процессов, связанных с заданием. Он также записывает основную учетную информацию для всех процессов, связанных с заданием, а также для всех процессов, которые были связаны с заданием ранее, но на данный момент уже завершены. Некоторым образом объект задания компенсирует в Windows отсутствие структурированного дерева процесса, но во многих отношениях он является более мощным средством, чем дерево процесса в UNIX-стиле.

Внутренняя структура заданий, процессов и потоков, механизмы создания процессов и потоков и алгоритмы планирования работы потоков более подробно рассмотрены в главе 5.

Виртуальная память

Windows реализует систему виртуальной памяти на основе плоского (линейного) адресного пространства, предоставляя каждому процессу иллюзию наличия его собственного, большого, закрытого адресного пространства. Виртуальная память предоставляет логическое представление памяти, которое может не соответствовать ее физическому расположению. Во время работы диспетчер памяти при содействии оборудования переводит, или отображает виртуальные адреса на физические, по которым и хранятся данные. Управляя защитой и отображением, операционная система может обеспечивать отсутствие столкновений процессов или перезаписи своих данных. На рис. 1.3 показаны три виртуально-последовательные страницы, отображенные на три непоследовательные страницы физической памяти.

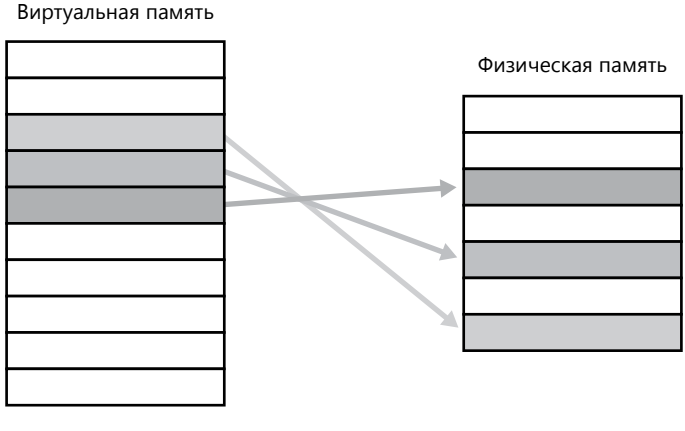


Рис. 1.3. Отображение виртуальной памяти на физическую

Поскольку у большинства систем физической памяти намного меньше общего количества виртуальной памяти, использующейся работающими процессами, диспетчер памяти переводит или осуществляет постраничный перенос части содержимого памяти на диск. Постраничный перенос данных на диск освобождает физическую память, чтобы она могла использоваться другими процессами или самой операционной системой. Когда поток обращается к памяти по виртуальному адресу той страницы, которая была перенесена на диск, диспетчер виртуальной памяти загружает информацию обратно с диска в память. Использование страничной подкачки не требует каких-либо изменений в приложениях, поскольку диспетчер памяти благодаря аппаратной поддержке справляется с этим, не ставя в известность процессы или потоки и не пользуясь их содействием.

Размер виртуального адресного пространства зависит от конкретной аппаратной платформы. На 32-разрядных системах x86 общее виртуальное адресное пространство имеет теоретический максимальный объем, равный 4 Гбайт. По умолчанию Windows распределяет половину этого адресного пространства (нижнюю половину 4-гигабайтного виртуального адресного пространства с адресами от 0x00000000 до 0x7FFFFFFF) между процессами для их уникальных закрытых хранилищ и использует другую половину (верхнюю, с адресами от 0x80000000 до 0xFFFFFFFF) в качестве своей собственной защищенной памяти операционной системы. Отображения нижней половины изменяются в соответствии с виртуальным адресным пространством текущих выполняемых процессов, а отображения верхней части всегда состоят из виртуальной памяти операционной системы. Windows поддерживает параметры загрузки[1], которые дают процессам, выполняющим специально помеченные программы[2], возможность использования до 3 Гбайт закрытого адресного пространства (оставляя 1 Гбайт для операционной системы). Это параметр позволяет таким приложениям, как серверы баз данных, хранить более крупные части баз данных в адресном пространстве процесса, сокращая тем самым потребности в отображении представлений подмножеств базы данных. На рис. 1.4 показаны две типичные схемы виртуальных адресных пространств, поддерживаемых 32-разрядной Windows. (Параметр increaseuserva позволяет помеченным приложениям использовать от 2 до 3 Гбайт адресного пространства.)

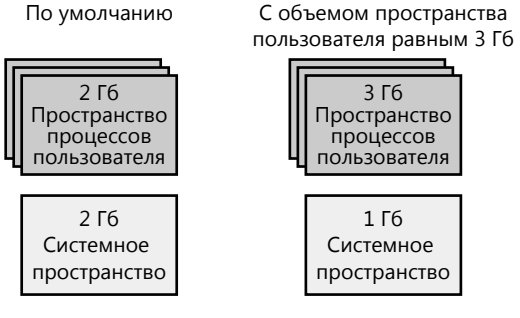


Рис. 1.4. Типичные схемы адресных пространств для 32-разрядной версии Windows

[1] Спецификатор increaseuserva в базе данных конфигурации загрузки — Boot Configuration Database.

[2] В заголовке исполняемого образа должен быть установлен флаг признака большого адресного пространства.

Несмотря на то что 3 Гбайт иметь лучше, чем 2 Гбайт, для отображения очень больших (многогигабайтных) баз данных этого объема все же не хватает. Чтобы отвечать их потребностям на 32-разрядных системах, Windows предоставляет механизм под названием Address Windowing Extension (AWE), позволяющий 32-разрядным приложениям выделять до 64 Гбайт физической памяти, а затем проецировать представления, или окна, на свое 2-гигабайтное виртуальное адресное пространство. Использование AWE возлагает на программиста обязанности по управлению отображением виртуальной памяти на физическую. Этот механизм удовлетворяет потребность в непосредственном доступе к большему объему физической памяти, чем тот, который может в любой заданный момент времени быть отображен на адресное пространство 32-разрядного процесса.

64-разрядная версия Windows предоставляет процессам намного более обширное адресное пространство: 7152 Гбайт на системах IA-64 и 8192 Гбайт на системах x64. На рис. 1.5 показано упрощенное представление структуры адресного пространства 64-разрядной системы. Следует заметить, что показанные объемы памяти не отображают архитектурных ограничений этих платформ. Шестьдесят четыре разряда адресного пространства позволяют адресовать более 17 миллиардов гигабайт, но имеющееся в настоящее время 64-разрядное оборудование ограничивает этот объем более скромным значением. А ограничения, связанные с реализацией текущей 64-разрядной операционной системы Windows, снижают доступное адресное пространство еще больше, сводя его к 8192 Гбайт (8 Тб).

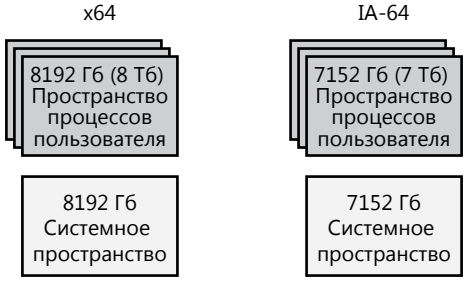


Рис. 1.5. Схемы адресных пространств для 64-разрядной версии Windows

Сравнение режима ядра и пользовательского режима

Чтобы защитить жизненно важные системные данные от доступа и (или) внесения изменений со стороны пользовательских приложений, в Windows используются два процессорных режима доступа (даже если процессор, на котором работает Windows, поддерживает более двух режимов): пользовательский режим и режим ядра. Код пользовательского приложения запускается в пользовательском режиме, а код операционной системы (например, системные службы и драйверы устройств) запускается в режиме ядра. Режим ядра — такой режим работы процессора, в котором предоставляется доступ ко всей системной памяти и ко всем инструкциям центрального процессора. Предоставляя программному обеспечению операционной системы более высокий уровень привилегий, нежели прикладному

программному обеспечению, процессор гарантирует, что приложения с неправильным поведением не смогут в целом нарушить стабильность работы системы.

ПРИМЕЧАНИЕ

В архитектурах процессоров x86 и x64 определены четыре уровня привилегий (или четыре кольца) для защиты системного кода и данных от непреднамеренной или злонамеренной перезаписи в результате выполнения кода, имеющего более низкий уровень привилегий. Windows использует уровень привилегий 0 (или кольцо 0) для режима ядра, и уровень привилегий 3 (или кольцо 3) для пользовательского режима. Причина, по которой в Windows используются только два уровня, заключается в том, что в некоторых аппаратных архитектурах, поддерживаемых в прошлом (например, Compaq Alpha и Silicon Graphics MIPS), были реализованы только два уровня привилегий.

Хотя у каждого Windows-процесса есть свое собственное закрытое адресное пространство, код операционной системы и код драйвера устройства, используют одно и то же общее виртуальное адресное пространство. Каждая страница в виртуальной памяти имеет пометку, показывающую, в каком режиме доступа должен быть процессор для чтения и (или) записи страницы. Доступ к страницам в системном пространстве может быть осуществлен только из режима ядра, тогда как доступ ко всем страницам в пользовательском адресном пространстве может быть осуществлен из пользовательского режима. Станицы, предназначенные только для чтения (например, те страницы, которые содержат статические данные), недоступны для записи из любого режима. Кроме того, при работе на процессорах, поддерживающих защиту той памяти, которая не содержит исполняемого кода (no-execute memory protection), Windows помечает страницы, содержащие данные, как неисполняемые, предотвращая тем самым неумышленное или злонамеренное выполнение кода из областей данных.

32-разрядные версии Windows не защищают закрытую системную память чтения-записи, используемую компонентами операционной системы, запущенными в режиме ядра. Иными словами, в режиме ядра код операционной системы и драйвера устройства имеют полный доступ к системному пространству памяти и могут обойти систему защиты Windows, получив доступ к объектам. Поскольку основная часть кода операционной системы Windows работает в режиме ядра, очень важно, чтобы компоненты, работающие в этом режиме, были тщательно проработаны и протестированы, чтобы не нарушать безопасность системы или не становиться причиной нестабильной работы системы.

Отсутствие защиты также подчеркивает необходимость проявлять особую осторожность при загрузке драйвера устройства стороннего производителя, потому что программное обеспечение, работающее в режиме ядра, имеет полный доступ ко всем данным операционной системы. Этот недостаток стал одной из причин введения в Windows механизма подписи драйверов, который выводит предупреждение пользователю при попытке добавления автоматически настраиваемого (Plug and Play) драйвера, не имеющего подписи (или, при определенной настройке, блокирует добавление такого драйвера). Помимо этого верификатор драйверов — Driver Verifier — помогает создателям драйверов выискивать просчеты (например, переполнение буферов или допущение утечек памяти), способные повлиять на безопасность или стабильность работы системы.

В 64-разрядных версиях Windows политика подписи кода в режиме ядра — Kernel Mode Code Signing (KMCS) — требует, чтобы все 64-разрядные драйверы устройств (не только автоматически настраиваемые) были подписаны криптографическим ключом, присвоенным одним из основных центров сертификации кода. Пользователь не может напрямую заставить систему установить неподписанный драйвер, даже имея права администратора, за единственным исключением: эти ограничения могут быть отключены вручную во время загрузки системы путем нажатия клавиши F8 и выбора дополнительного параметра загрузки Disable Driver Signature Enforcement (Выключить принуждение к подписыванию драйверов). При этом выключаются водяной знак на обоях для рабочего стола и определенные функции системы управления правами на цифровые материалы — digital rights management (DRM).

В главе 2 «Архитектура системы» будет показано, что пользовательские приложения осуществляют переключение из пользовательского режима в режим ядра при осуществлении вызова системной службы. Например, Windows-функции ReadFile, в конечном счете, необходим вызов внутренней стандартной программы Windows, управляющей чтением данных из файла. Поскольку эта стандартная программа обращается к структурам внутренних системных данных, она должна работать в режиме ядра. Переход из режима пользователя в режим ядра осуществляется за счет использования специальной инструкции процессора, которая заставляет процессор переключиться в режим ядра и войти в код диспетчеризации системных служб, вызывающий соответствующую внутреннюю функцию в Ntoskrnl.exe или в Win32k.sys. Перед тем как вернуть управление пользовательскому потоку, процессор переключается в прежний, пользовательский режим работы. Таким образом, операционная система защищает саму себя и свои данные от прочтения и модификации со стороны пользовательских процессов.

ПРИМЕЧАНИЕ

Переход из пользовательского режима в режим ядра (и назад) не влияет на планирование работы потоков как таковое — переход из режима в режим не является переключением контекста. Более подробно диспетчеризация системных служб рассматривается в главе 3.

Таким образом, пользовательский поток вполне может выполняться часть времени в пользовательском режиме, а другую часть времени — в режиме ядра. Фактически, из-за того что основная масса графики и оконная система также работают в режиме ядра, приложения, интенсивно использующие графику, проводят большую часть своего времени в режиме ядра, нежели в пользовательском режиме. Это легко проверить, если запустить приложение, интенсивно использующее графику, например Microsoft Paint или Microsoft Chess Titans, и посмотреть, как распределяется время между пользовательским режимом и режимом ядра, используя для этого один из счетчиков производительности, перечисленных в табл. 1.2. Более сложные приложения могут использовать такие новые технологии, как Direct2D и создание составных изображений (compositing), которые проводят основной объем вычислений в пользовательском режиме и отправляют ядру только исходные данные поверхностей, сокращая время, затрачиваемое на переходы между пользовательскими режимами и режимами ядра.

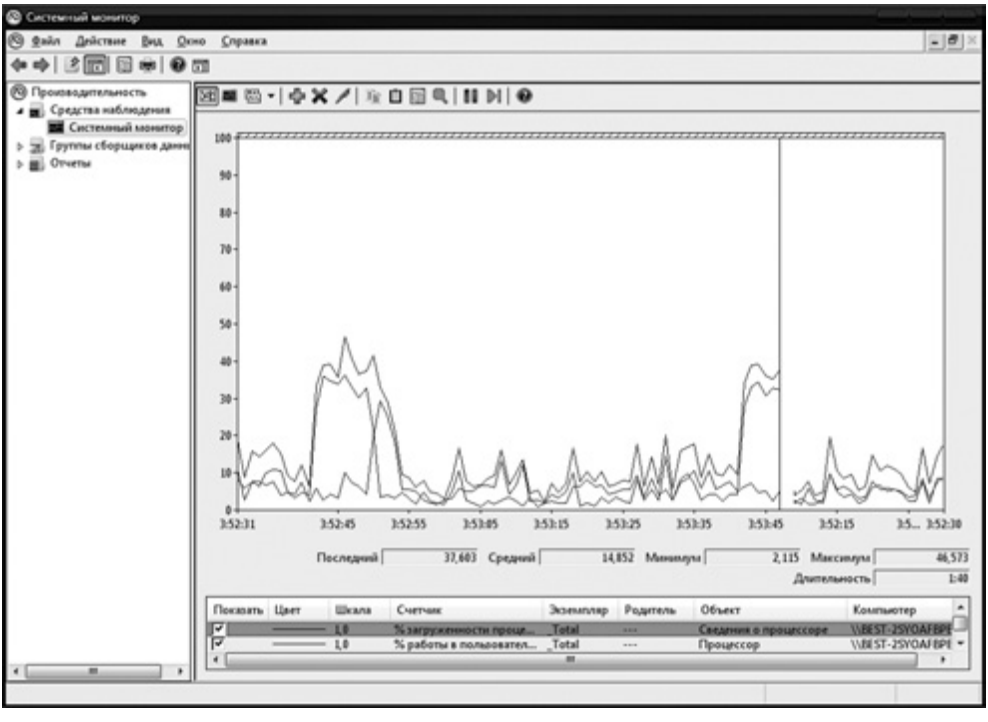
Таблица 1.2. Счетчики производительности, связанные с режимами работы процессора

Объект: Счетчик	Функция
Процессор: % работы в привилегированном режиме (Processor: % Privileged Time)	Процентный показатель работы отдельного центрального процессора (или всех центральных процессоров) в режиме ядра в течение определенного интервала времени
Процессор: % работы в пользовательском режиме (Processor: % User Time)	Процентный показатель отдельного центрального процессора (или всех центральных процессоров) в пользовательском режиме в течение определенного интервала времени
Процесс: % работы в привилегированном режиме (Process: % Privileged Time)	Процентный показатель работы потоков процесса в режиме ядра в течение определенного интервала времени
Процесс: % работы в пользовательском режиме (Process: % User Time)	Процентный показатель работы потоков процесса в пользовательском режиме в течение определенного интервала времени
Поток: % работы в привилегированном режиме (Thread: % Privileged Time)	Процентный показатель работы потока в режиме ядра в течение определенного интервала времени
Поток: % работы в пользовательском режиме (Thread: % User Time)	Процентный показатель работы потока в пользовательском режиме в течение определенного интервала времени

ЭКСПЕРИМЕНТ: СРАВНЕНИЕ ВРЕМЕНИ РАБОТЫ В РЕЖИМА ЯДРА И В ПОЛЬЗОВАТЕЛЬСКОМ РЕЖИМЕ

Чтобы посмотреть, сколько времени ваша система работает в режиме ядра по сравнению с работой в пользовательском режиме, можно воспользоваться Системным монитором (Performance Monitor). Выполните следующие действия:

1. Запустите Системный монитор (Performance Monitor), открыв меню Пуск (Start) и выбрав пункты Панель управления ▸ Администрирование ▸ Системный монитор (All Programs ▸ Administrative Tools ▸ Performance Monitor). На расположенном слева древовидном раскрывающемся списке инструментов Производительность (Performance) выберите пункты Средства наблюдения (Monitoring Tools)▸ Системный монитор (Performance Monitor).
2. Щелкните на кнопке добавления (+), которая находится на панели инструментов.
3. Раскройте раздел счетчиков Процессор (Processor), щелкните на пункте % работы в привилегированном режиме (% Privileged Time counter) и, удерживая в нажатом состоянии клавишу Ctrl, щелкните на пункте % работы в пользовательском режиме (% User Time).
4. Щелкните на кнопке Добавить (Add), а затем на кнопке OK.
5. Откройте окно командной строки и проведите непосредственное сканирование своего диска C по сети, набрав команду dir \\%computername%\c$ /s.



6. По окончании работы закройте окно инструментального средства.

Такую же картину можно быстро просмотреть с помощью Диспетчера задач (Task Manager). Щелкните на вкладке Производительность (Performance), а затем выберите в меню Вид (View) пункт Вывод времени ядра (Show Kernel Times). На графике загруженности центрального процессора зеленым цветом будет показана его общая загруженность, а красным — загруженность в режиме ядра.

Чтобы увидеть, сколько времени в режиме ядра и в пользовательском режиме использует сам Системный монитор (Performance Monitor), запустите его еще раз, но при этом добавьте отдельные счетчики процесса % работы в пользовательском режиме (% User Time) и % работы в привилегированном режиме (% Privileged Time) для каждого процесса в системе:

1. Если Системный монитор (Performance Monitor) не запущен, запустите его снова. (Если он уже запущен, начните работу с пустого отображения, щелкнув в области графиков правой кнопкой мыши и выбрав пункт Удалить все счетчики (Remove All Counters.)
2. Щелкните на кнопке добавления (+), которая находится на панели инструментов.
3. В доступной области счетчиков раскройте раздел Процесс (Process).
4. Выберите счетчики % работы в пользовательском режиме (% User Time) и % работы в привилегированном режиме (% Privileged Time).
5. Выберите несколько процессов в области Экземпляры выбранного объекта (Instance) (например, mmc, csrss и Idle).
6. Щелкните на кнопке Добавить (Add), а затем на кнопке OK.

7. Интенсивно подвигайте мышью в разные стороны.
8. Выберите на панели инструментов пункт Выделить (Highlight) или нажмите сочетание клавиш Ctrl+H, чтобы включить режим выделения. Текущий выбранный счетчик будет выделен черным цветом.
9. Прокрутите список счетчиков вниз для определения процессов, чьи потоки были запущены при перемещении указателя мыши, и обратите внимание на то, в каком режиме они были запущены, в пользовательском или в режиме ядра.

Вы должны увидеть (найдя в столбце Экземпляр (Instance) процесс mmc), что график времени выполнения процесса, принадлежащего Системному монитору, в режиме ядра и в пользовательском режиме при перемещении мыши пошел вверх, поскольку в нем выполняется прикладной код в пользовательском режиме, и вызываются Windows-функции, запускаемые в режиме ядра. Обратите также внимание на активность потока, принадлежащего процессу csrss и выполняемого в режиме ядра при перемещении мыши. Эта активность возникает благодаря тому, что этому процессу принадлежит исходный поток ввода той подсистемы Windows, выполняемой в режиме ядра, которая обрабатывает ввод с клавиатуры и с мыши. (Более подробная информация о системных потоках дается в главе 2.) И наконец, процесс Idle, который, как можно заметить, тратит почти 100 % своего времени на работу в режиме ядра, на самом деле процессом не является, это ложный процесс, используемый для подсчета холостых циклов центрального процессора. Судя по режиму, в котором запускаются потоки процесса Idle, когда Windows нечего делать, процесс ожидания происходит в режиме ядра. ■

Службы терминалов и множественные сеансы работы

Службы терминалов (Terminal Services)[1] относятся к поддержке в Windows нескольких интерактивных сеансов работы пользователей на одной и той же системе. С помощью этих служб удаленный пользователь может установить сеанс работы на другой машине, войти в систему и запустить приложения на сервере. Сервер передает клиенту графический интерфейс пользователя (а также другие настраиваемые ресурсы, такие как управление звуковой подсистемой и буфером обмена), а клиент передает обратно на сервер пользовательский ввод. (Как и X Window System, Windows разрешает запуск отдельных приложений с удаленным отображением на стороне клиента, вместо удаленного взаимодействия со всем рабочим столом.)

Первый сеанс считается сеансом служб, или сеансом нуль (session zero), и содержит процессы, в которых реализуются системные службы (более подробно этот сеанс рассматривается в главе 4 «Механизмы управления»). Первый сеанс входа в систему на физической консоли машины является сеансом номер один, а дополнительные сеансы могут создаваться с помощью программы подключения к удаленному рабочему столу (Mstsc.exe) или с использованием быстрого переключения пользователей, которое мы рассмотрим чуть позже.

Клиентские версии Windows допускают подключение к машине одного удаленного пользователя, но если кто-нибудь зарегистрируется в консоли, рабочая станция блокируется (то есть системой можно пользоваться либо в локальном, либо

[1] В последних версиях Windows — Службы удаленных рабочих столов. — *Примеч. перев.*

в удаленном режиме, но одновременно в обоих режимах ею пользоваться нельзя). Версии Windows, включающие Windows Media Center, допускают проведение одного интерактивного сеанса и до четырех сеансов Windows Media Center Extender.

Серверные системы Windows поддерживают два одновременных удаленных подключения (для содействия удаленному управлению, например для использования средств управления, требующих регистрации на управляемой машине) и более двух удаленных сеансов, если серверные системы соответствующим образом лицензированы и настроены в качестве терминального сервера.

Все клиентские версии поддерживают несколько созданных локально сеансов, которые могут использоваться поочередно с помощью функции, называемой быстрым переключением пользователей. Когда пользователь выбирает вместо выхода из сеанса отключение своего сеанса (например, щелчком на кнопке Пуск (Start) с последующим выбором пункта Сменить пользователя (Switch User) из подменю Завершение работы (Shutdown) или удержанием в нажатом состоянии клавиши Windows с последующим нажатием клавиши L и щелчком на кнопке Сменить пользователя (Switch User button)), текущий сеанс (процесс, запущенный в этом сеансе, и все, относящиеся к этому процессу структуры данных, дающих описание сеанса) остаются активными в системе, и система возвращается к основному экрану входа в систему. Если в системе регистрируется новый пользователь, создается новый сеанс.

Для приложений, которым нужно знать об их запуске в сеансе терминального сервера, есть набор интерфейсных функций Windows API для определения этого факта программным путем, а также для управления различными аспектами служб терминалов. (Подробности можно найти в документации по Windows SDK и по Remote Desktop Services API.)

В главе 2 дается краткое описание порядка создания сеансов и приводится ряд экспериментов, показывающих, как просматривать информацию о сеансах с помощью различных инструментальных средств, включая отладчик ядра. В главе 3 в разделе «Диспетчер объектов» рассматривается вопрос создания экземпляров системного пространства имен для объектов на сеансовой основе и вопрос определения приложениями, если это им необходимо, наличия других своих экземпляров на той же самой системе.

Объекты и дескрипторы

В операционной системе Windows объект ядра является единственным экземпляром типа объекта времени выполнения, определенного в статическом режиме. Тип объекта состоит из определяемого системой типа данных, функций (методов), работающих с экземплярами типа объекта и набора свойств объекта. При написании Windows-приложений могут встретиться объекты процессов, потоков, файлов и событий (и это лишь несколько примеров). Эти объекты основаны на низкоуровневых объектах, создаваемых и управляемых операционной системой Windows, в которой процесс является экземпляром типа объекта process, файл является экземпляром типа объекта file и т. д.

Свойство объекта является находящимся в объекте полем данных, которое в той или иной степени определяет состояние объекта. К примеру, у объекта типа process будут свойства, включающие идентификатор процесса (process ID),

основной приоритет, учитываемый при планировании запуска процесса, и указатель на объект маркера доступа. Методы объекта, то есть средства манипуляции объектами, обычно считывают или изменяют свойства объекта. Например, методу open объекта process в качестве входных данных будет передаваться идентификатор процесса, а на выходе он будет возвращать указатель на объект.

ПРИМЕЧАНИЕ

Хотя при создании объекта с использованием API диспетчера объектов ядра вызывающий код предоставляет аргумент ObjectAttributes (свойства объекта), этот аргумент не нужно путать с используемым в данной книге более общим значением термина «свойства объекта».

Наиболее существенным отличием объекта от обычной структуры данных является недоступность внутренней структуры объекта за его пределами. Для извлечения данных из объекта или для помещения данных в объект нужно вызвать службу объекта. Напрямую прочитать или изменить данные внутри объекта просто невозможно. Это отличие отделяет базовую реализацию объекта от того кода, который его просто использует. Благодаря такой технологии реализацию объектов впоследствии будет нетрудно изменить.

Объекты с помощью компонента ядра под названием «диспетчер объектов» предоставляют удобные средства для выполнения следующих четырех важных задач операционной системы:

- Предоставление легких для человеческого восприятия имен системных ресурсов.
- Распределение ресурсов и данных среди процессов.
- Защита ресурсов от неавторизованного доступа.
- Отслеживание ссылок, позволяющее системе узнать, когда объект больше не используется, чтобы можно было автоматически освободить выделенные под него ресурсы.

Но в операционной системе Windows объектами являются не все структуры данных. В объекты помещаются только те данные, которые нужно использовать совместно, защитить, снабдить именами или сделать видимыми (через системные службы) для программ, выполняемых в пользовательском режиме. Структуры, используемые только одним компонентом операционной системы для реализации внутренних функций, объектами не являются. Объекты и дескрипторы (ссылки на экземпляр объекта) рассматриваются в главе 3.

Безопасность

С самого начала Windows разрабатывалась с прицелом на безопасность и на соответствие различным официальным правительственным и промышленным требованиям к безопасности, таким как спецификация под названием «Общие критерии оценки безопасности информационных систем» — Common Criteria for Information Technology Security Evaluation (CCITSE). Достижение уровней безопасности, утвержденных правительством, позволяет операционной системе быть конкурентоспособной в сфере правительственных закупок. Разумеется,

многие из соответствующих этим требованиям возможностей являются предпочтительными свойствами для любой многопользовательской системы.

Основные возможности в сфере обеспечения безопасности Windows включают в себя:

- защиту, предоставляемую на усмотрение (узкого круга лиц), а также обязательную защиту целостности для всех общих системных объектов (файлов, каталогов, процессов, потоков и т. д.);
- контроль безопасности (для возможности идентификации субъектов, или пользователей, и инициированных ими действий);
- аутентификацию пользователей при входе в систему и предотвращение доступа одного пользователя к неинициализированным ресурсам (таким как свободное пространство памяти или дисковое пространство), освобожденным другим пользователем.

В Windows имеются три формы управления доступом к объектам. Первая форма — управление избирательным доступом (discretionary access control) — представляет собой защитный механизм, который многими и воспринимается в качестве системы безопасности операционной системы. Этот метод позволяет владельцам объектов (таких как файлы и принтеры) предоставлять доступ или отказывать в доступе всем остальным пользователям. При входе пользователей в систему им дается набор прав доступа, или контекст безопасности. При попытке доступа к объектам их контекст безопасности сравнивается со списком управления доступом (access control list) объекта, к которому осуществляется попытка доступа, чтобы определить, есть ли у того или иного пользователя права на осуществление запрошенной операции.

Когда управления избирательным доступом недостаточно, необходимо управление привилегированным доступом (privileged access control). Оно представляет собой метод, гарантирующий чей-либо доступ к защищенным объектам в отсутствие их владельца. Например, если работник уходит из компании, администратору нужен способ получения доступа к файлам, которые могут быть доступны только этому работнику. В случае когда работа ведется под управлением Windows, администратор может стать владельцем файла и, если это необходимо, распорядиться правами доступа к этому файлу.

И наконец, когда требуется дополнительный уровень безопасности для защиты тех объектов, которые доступны в пределах прав одной и той же учетной записи пользователя, нужен обязательный контроль целостности. Он используется как для изоляции браузера Internet Explorer, работающего в защищенном режиме, от пользовательской конфигурации, так и для защиты объектов, созданных при работе с правами учетной записи администратора с расширенными привилегиями, от доступа к ним со стороны пользователей, работающих с правами учетной записи администратора, не имеющего расширенных привилегий. (Более подробная информация об управлении учетными записями пользователей — User Account Control, UAC, дана в главе 6.)

Безопасность проникает и в интерфейс Windows API. Безопасность на основе объектов реализуется в подсистемах Windows точно так же, как это делается в самой операционной системе. Подсистемы Windows защищают общие Windows-объекты от неавторизованного доступа путем размещения в них дескрипторов безопасности.

При первой попытке приложения получить доступ к общему объекту подсистема Windows проверяет наличие у приложения прав на подобное действие. Если проверка безопасности проходит успешно, подсистема Windows позволяет приложению продолжить свое действие.

Более полное описание системы безопасности Windows дано в главе 6.

Реестр

Если вам приходилось работать с операционными системами Windows, вы, наверное, уже слышали о реестре или даже видели его. Нельзя говорить о внутреннем устройстве Windows, не упоминая при этом реестр, поскольку он представляет собой системную базу данных. В реестре содержится информация, необходимая для загрузки и настройки системы, общесистемных программных настроек, управляющих работой Windows, настроек базы данных безопасности и настроек конфигурации конкретного пользователя (например, какую заставку использовать).

Кроме того, реестр является своеобразным окном в запомненные непостоянные данные — например, данные, касающиеся текущего состояния оборудования системы (какие драйверы устройств загружены, каковы используемые ими ресурсы и т. д.), а также в счетчики производительности Windows. Эти счетчики, которые на самом деле находятся не в реестре, доступны через функции реестра. Более подробно о том, как из реестра можно получить доступ к информации счетчика производительности, рассказывается в главе 4.

Хотя многим пользователям и администраторам Windows заглядывать непосредственно в реестр может никогда и не понадобиться (просматривать или вносить изменения в большинство настроек конфигурации можно с помощью стандартных программ администрирования.), он все равно является полезным источником внутренней информации Windows, потому что в нем содержатся многие настройки, влияющие на производительность и поведение системы. Решив напрямую внести изменения в параметры реестра, нужно проявлять предельную осторожность, любые изменения могут плохо отразиться на производительности системы или, что еще хуже, помешать успешной загрузке системы. В данной книге будут встречаться ссылки на отдельные разделы реестра, относящиеся к рассматриваемым компонентам. Большинство разделов, на которые даются ссылки в данной книге, относятся к общесистемной конфигурации, находящейся в разделе HKEY_LOCAL_MACHINE, для которого будет использоваться аббревиатура HKLM.

Дополнительная информация о реестре и его внутренней структуре дается в главе 4.

Unicode

Windows отличается от других операционных систем тем, что большинство внутренних текстовых строк в ней хранится и обрабатывается в виде расширенных 16-разрядных символьных кодов Unicode. По своей сути Unicode является стандартом международных наборов символов, определяющим 16-разрядные значения для наиболее известных во всем мире наборов символов.

Поскольку многие приложения работают со строками, состоящими из 8-разрядных (однобайтовых) ANSI-символов, многие Windows-функции, которым передаются строковые параметры, имеют две точки входа: в версии Unicode

(расширенной, 16-разрядной) и в версии ANSI (узкой, 8-разрядной). При вызове узкой версии Windows-функции происходит небольшое снижение производительности, поскольку входящие строковые аргументы перед обработкой преобразуются в Unicode, а после обработки, при возвращении приложению, проходят обратное преобразование из Unicode в ANSI. Поэтому если у вас есть старая служба или старый фрагмент кода, который нужно запустить под управлением Windows, но этот фрагмент создан и с использованием текстовых строк, состоящих из ANSI-символов, Windows для своего собственного использования преобразует ANSI-символы в Unicode. Но Windows никогда не станет конвертировать данные внутри файлов: решение о том, в какой кодировки хранить данные: в Unicode или в ANSI, принимается самим приложением.

Независимо от языка, во всех версиях Windows содержатся одни и те же функции. Вместо использования отдельных версий для каждого языка, в Windows используется единый универсальный двоичный код, поэтому отдельно взятая установка может поддерживать несколько языков (путем добавления различных языковых пакетов). Приложения могут также воспользоваться Windows-функциями, предоставляющими возможность во всем мире использовать одни и те же исполняемые файлы приложений, поддерживающие сразу несколько языков.

Дополнительную информацию о Unicode можно найти на сайте www.unicode.org, а также в документации по программированию в библиотеке MSDN.

Подробное исследование внутреннего устройства Windows

Хотя основная масса информации в этой книге основана на чтении исходного кода Windows и на разговорах с разработчиками, вы не должны принимать все на веру. Многие подробности внутреннего устройства Windows могут быть обнаружены и продемонстрированы с помощью использования множества доступных инструментальных средств, поставляемых с Windows или со средствами отладки, работающими под управлением Windows. В данном разделе мы кратко рассмотрим эти инструментальные пакеты.

Чтобы пробудить в вас интерес к исследованию внутреннего устройства Windows, по всей книге разбросаны врезки «Эксперимент», в которых описываются действия, которые нужно выполнить для изучения того или иного аспекта внутреннего поведения Windows. Мы рекомендуем вам провести эти эксперименты, чтобы посмотреть в действии многие вопросы внутреннего устройства Windows, рассматриваемые в данной книге.

В табл. 1.3 приведен перечень основных инструментальных средств, используемых в данной книге, и указаны источники их поступления.

Таблица 1.3. Инструментальные средства для просмотра внутреннего устройства Windows

Инструментальное средство	Имя образа	Источник поступления
Startup Programs Viewer (Средство для просмотра программ, автоматически запускаемых при загрузке системы)	AUTORUNS	Sysinternals[1]

[1] http://technet.microsoft.com/ru-RU/sysinternals. — *Примеч. перев.*

Инструментальное средство	Имя образа	Источник поступления
Access Check (Средство для проверки прав доступа)	ACCESSCHK	Sysinternals
Dependency Walker (Средство для обхода зависимостей)	DEPENDS	www.dependencywalker.com
Global Flags (Средство для работы с глобальными флагами)	GFLAGS	Средства отладки
Handle Viewer (Средство для вывода сведений об открытых дескрипторах для любого процесса в системе)	HANDLE	Sysinternals
Kernel debuggers (Средство отладки ядра)	WINDBG, KD	Средства отладки, Windows SDK
Object Viewer (Средство для просмотра объектов)	WINOBJ	Sysinternals
Performance Monitor (Системный монитор)	PERFMON.MSC	Встроенное средство Windows
Pool Monitor (Монитор пула памяти)	POOLMON	Windows Driver Kit
Process Explorer (Средство для исследования процессов)	PROCEXP	Sysinternals
Process Monitor (Средство для отслеживания процессов)	PROCMON	Sysinternals
Task (Process) List (Средство для вывода перечня задач (процессов))	TLIST	Средства отладки
Task Manager (Диспетчер задач)	TASKMGR	Встроенное средство Windows

Системный монитор

Ссылки на Системный монитор (Performance Monitor), доступный через Панель управления (Control Panel), будут встречаться по всей книге. Особое внимание будет уделяться Системному монитору (Performance Monitor) и Монитору ресурсов (Resource Monitor). Системный монитор выполняет три функции: мониторинг системы, просмотр журналов счетчиков производительности и настройка оповещений (путем использования настроек сборщика данных, который также содержит журналы и трассировку счетчиков производительности и настроечные данные).

Системный монитор предоставляет больше информации о работе вашей системы, чем любое другое отдельно взятое средство. Он включает в себя сотни основных и расширенных счетчиков для различных объектов. Для каждой основной темы, рассматриваемой в данной книге, включается таблица самых важных счетчиков производительности Windows.

В Системном мониторе содержится краткое описание каждого счетчика. Чтобы увидеть описания, нужно в окне Добавить счетчики (Add Counters) установить флажок Отображать описание (Show Description).

Хотя весь низкоуровневый системный мониторинг, рассматриваемый в данной книге, может проводиться с помощью Системного монитора, Windows

также включает служебную программу Монитор ресурсов (запускается из меню Пуск или из вкладки Быстродействие (Performance) Диспетчера задач (Task Manager)), которая показывает четыре основных ресурса: центральный процессор, диск, сеть и память. В своих основных состояниях эти ресурсы показываются с тем же уровнем информации, который можно найти в Диспетчере задач. Но к этому добавляются области, которые могут быть развернуты для получения дополнительной информации.

При раскрытии вкладки ЦП (CPU) показывается информация об использовании центрального процессора для каждого процесса, точно так же, как в Диспетчере задач. Но в этой вкладке добавлен столбец для среднего показателя использования центрального процессора, который может дать более наглядное представление о том, какой из процессоров наиболее активен. Во вкладку ЦП (CPU) также включается отдельное отображение служб, используемого ими центрального процессора и среднего показателя использования этого процессора. Каждый процесс, в рамках которого выполняется служба (хост-процесс), идентифицируется группой той службы, которая на нем выполняется. Как и при использовании Process Explorer, выбор процесса (путем установки соответствующего флажка) приведет к отображению списка поименованных дескрипторов, открытых процессом, а также списка модулей (например, DLL-библиотек), загруженных в адресное пространство процесса. Поле Поиск дескрипторов (Search Handles) может также использоваться для поиска тех процессов, которые открыли дескриптор для заданного поименованного ресурса.

В разделе Память (Memory) отображается почти такая же информация, которую можно получить с помощью Диспетчера задач, но она упорядочена в отношении всей системы. Гистограмма физической памяти отображает текущую организацию этой памяти, разбивая весь объем памяти на зарезервированную, используемую, измененную, находящуюся в режиме ожидания и свободную.

В разделе Диск (Disk), в отличие от остальных разделов, пофайловая информация для ввода-вывода отображается таким образом, чтобы было проще определить наиболее востребованные по записи или чтению файлы системы. Эти результаты могут подвергаться дальнейшей фильтрации по процессам.

В разделе Сеть (Networking) отображаются активные сетевые подключения и владеющие ими процессы, а также количество данных, прошедшее через эти подключения. Эта информация дает возможность увидеть фоновую сетевую активность, которую другим способом может быть трудно обнаружить. Кроме этого показываются имеющиеся в системе активные TCP-подключения, упорядоченные по процессам, с демонстрацией таких данных, как удаленный и локальный порт и адрес, и задержка пакета. И наконец, отображается список прослушиваемых процессом портов, позволяющий администратору увидеть, какие службы (или приложения) в данный момент ожидают подключения к тому или иному порту. Также показываются протокол и политика брандмауэра для каждого порта и процесса.

Следует заметить, что все счетчики производительности Windows являются программно доступными. В разделе «HKEY_PERFORMANCE_DATA» главы 4 дается краткое описание компонентов, используемых для извлечения показаний счетчиков производительности с помощью Windows API.

Отладка ядра

Термин «отладка ядра» означает изучение внутренней структуры данных ядра и (или) пошаговую трассировку функций в ядре. Эта отладка является весьма полезным способом исследования внутреннего устройства Windows, поскольку она позволяет получить отображения внутренней системной информации, недоступной при использовании каких-либо других средств, и дает четкое представление о ходе выполнения кода в ядре.

Прежде чем рассматривать различные способы отладки ядра, давайте исследуем набор файлов, который понадобится для осуществления любого вида такой отладки.

Символы для отладки ядра

Файлы символов содержат имена функций и переменных, а также схему и формат структур данных. Они генерируются программой-компоновщиком (linker) и используются отладчиками для ссылок на эти имена и для их отображения во время сеанса отладки. Эта информация обычно не хранится в двоичном коде, поскольку при выполнении кода она не нужна. Это означает, что без нее двоичный код становится меньше по размеру и выполняется быстрее. Но это также означает, что при отладке нужно обеспечить отладчику доступ к файлам символов, связанных с двоичными образами, на которые идут ссылки во время сеанса отладки.

Для использования любого средства отладки в режиме ядра с целью исследования внутреннего устройства структуры данных ядра Windows (списка процессов, блоков потоков, списка загруженных драйверов, информации об использовании памяти и т. д.) вам нужны правильные файлы символов и, как минимум, файл символов для двоичного образа ядра — Ntoskrnl.exe. (Более подробно этот файл рассматривается в разделе «Краткий обзор архитектуры» главы 2.) Файлы таблицы символов должны соответствовать версии того двоичного образа, из которого они были извлечены. Например, если установлен пакет Windows Service Pack или какое-нибудь исправление, обновляющее ядро, нужно получить соответствующим образом обновленные файлы символов.

Загрузить и установить символы для различных версий Windows нетрудно, а вот обновить символы для исправлений удается не всегда. Проще всего получить нужную версию символов для отладки путем обращения к специально предназначенному для этого серверу символов Microsoft, воспользовавшись для этого специальным синтаксисом для пути к символам, указываемом в отладчике. Например, следующий путь к символам заставляет средства отладки загрузить символы с интернет-сервера символов и сохранить локальную копию в папке c:\symbols:

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

Подробные инструкции по использованию символьного сервера можно найти в файле справки средств отладки или в Интернете на веб-странице http://msdn.microsoft.com/en-us/windows/hardware/gg462988.aspx.

Средства отладки для Windows

В пакет средств отладки для Windows входят современные отладочные инструменты, с помощью которых в данной книге предлагается исследовать внутреннее

устройство Windows. В самую последнюю версию в качестве составной части включен набор для разработки программного обеспечения — Windows Software Development Kit (SDK). Средства из этого набора могут использоваться для отладки как процессов пользовательского режима, так и процессов ядра. (См. соответствующую врезку.)

ПРИМЕЧАНИЕ

Средства отладки Debugging Tools for Windows довольно часто обновляются и выпускаются независимо от версий операционной системы Windows, поэтому почаще проверяйте наличие новых версий.

СОВЕТ. ОТЛАДКА В ПОЛЬЗОВАТЕЛЬСКОМ РЕЖИМЕ

Средства отладки могут также использоваться для подключения к процессу пользовательского режима и для изучения и (или) изменения состояния памяти процесса. При подключении к процессу есть два варианта:

- Навязчивый (Invasive). Если при подключении к запущенному процессу не даны специальные указания, для подключения отладчика к отлаживаемому коду используется Windows-функция DebugActiveProcess. Тем самым создаются условия для исследования и (или) изменения памяти процесса, установки контрольных точек и выполнения других отладочных функций. Windows позволяет остановить отладку без прерывания целевого процесса, если отладчик отключается без прерывания своей работы.
- Ненавязчивый (Noninvasive). При таком варианте отладчик просто открывает процесс с помощью функции OpenProcess. Этот процесс не подключается к другому процессу в качестве отладчика. Это позволяет исследовать и (или) изменять память целевого процесса, но вы не можете устанавливать контрольные точки.

Со средствами отладки можно также открывать файлы дампа процесса пользовательского режима (см. главу 3, раздел, посвященный диспетчеризации исключений).

Для отладки ядра могут использоваться два отладчика: работающий в окне командной строки (Kd.exe) и имеющий графический пользовательский интерфейс, GUI (Windbg.exe). Оба отладчика предоставляют одинаковый набор команд, поэтому выбор всецело зависит от личных предпочтений. Эти средства позволяют выполнять три типа отладки ядра:

- Открыть аварийный дамп-файл, созданный в результате аварийного завершения работы системы.
- Подключиться к действующей, работающей системе и исследовать состояние системы (или установить контрольные точки, если ведется отладка кода драйвера устройства). Для этой операции нужны два компьютера — целевой и ведущий. Целевой компьютер содержит отлаживаемую систему, а ведущий — систему, на которой запущен отладчик. Целевая система может быть подключена к ведущей через нуль-модемный кабель, кабель IEEE 1394 или отладочный кабель USB 2.0. Целевая система должна быть загружена в ре-

жиме отладки (либо путем нажатия F8 в процессе загрузки и выбора пункта Режим отладки (Debugging Mode), либо путем настройки системы на запуск в режиме отладки, используя Bcdedit или Msconfig.exe). Можно также подключиться через поименованный канал, применяемый при отладке через виртуальную машину (созданную такими средствами, как Hyper-V, Virtual PC или VMWare), путем выставления гостевой операционной системой последовательного порта в качестве поименованного канального устройства.

- Системы Windows позволяют также подключиться к локальной системе и исследовать ее состояние. Это называется «локальной отладкой ядра». Чтобы приступить к локальной отладке ядра с помощью отладчика WinDbg, откройте меню File (Файл), выберите пункт Kernel Debug (Отладка ядра), щелкните на вкладке Local (Локальная), а затем щелкните на кнопке OK. Целевая система должна быть загружена в отладочном режиме. Пример появляющегося при этом экрана показан на рис. 1.6. В режиме локальной отладки ядра не работают некоторые команды отладчика ядра (например, команда .dump, предназначенная для создания дампа памяти, хотя такой дамп можно создать с помощью рассматриваемого далее средства LiveKd).

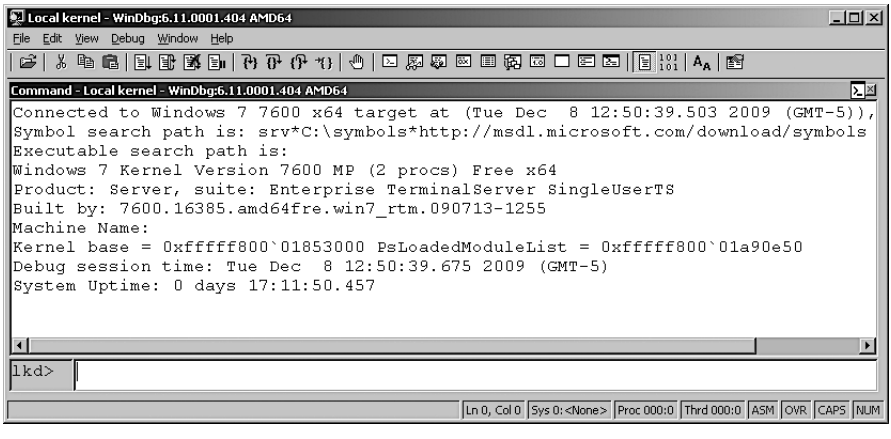


Рис. 1.6. Локальная отладка ядра

Для отображения содержимого внутренней структуры данных, включающей сведения о потоках, процессах, пакетах запросов на ввод-вывод данных и информации об управлении памятью, после подключения к режиму отладки ядра можно воспользоваться одной из множества команд расширения отладчика (команд, начинающихся с символа «!»). По каждой рассматриваемой теме в материал данной книги будут включаться соответствующие команды отладки ядра и состояния экрана отладчика. Отличным подсобным справочным материалом может послужить файл Debugger.chm, содержащийся в установочной папке отладчика WinDbg. В нем приводится документация по всем функциональным возможностям и расширениям отладчика ядра. В дополнение к этому команда dt (display type — отобразить тип) может отформатировать свыше 1000 структур ядра, поскольку в файлах символов ядра для Windows содержится информация о типах, которые отладчик может использовать для форматирования структур.

ЭКСПЕРИМЕНТ: ОТОБРАЖЕНИЕ ИНФОРМАЦИИ О ТИПАХ ДЛЯ СТРУКТУР ЯДРА

Чтобы вывести список структур ядра, чей тип информации включен в символы ядра, наберите в отладчике ядра команду dt nt!_*. Частичный образец вывода имеет следующий вид:

```
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
```

Командой dt можно также воспользоваться для поиска определенных структур, используя заложенную в эту команду возможность применения символов-заместителей. Например, если ведется поиск имени структуры для объекта interrupt, нужно набрать команду dt nt!_*interrupt*:

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

Затем, как показано в следующем примере, команду dt можно использовать для форматирования определенной структуры:

```
lkd> dt nt!_kinterrupt
nt!_KINTERRUPT
   +0x000 Type                   : Int2B
   +0x002 Size                   : Int2B
   +0x008 InterruptListEntry     : _LIST_ENTRY
   +0x018 ServiceRoutine         : Ptr64     unsigned char
   +0x020 MessageServiceRoutine : Ptr64     unsigned char
   +0x028 MessageIndex           : Uint4B
   +0x030 ServiceContext         : Ptr64 Void
   +0x038 SpinLock               : Uint8B
   +0x040 TickCount              : Uint4B
   +0x048 ActualLock             : Ptr64 Uint8B
   +0x050 DispatchAddress        : Ptr64     void
   +0x058 Vector                 : Uint4B
   +0x05c Irql                   : UChar
   +0x05d SynchronizeIrql        : UChar
   +0x05e FloatingSave           : UChar
   +0x05f Connected              : UChar
   +0x060 Number                 : Uint4B
   +0x064 ShareVector            : UChar
   +0x065 Pad                    : [3] Char
   +0x068 Mode                   : _KINTERRUPT_MODE
```

```
   +0x06c Polarity         : _KINTERRUPT_POLARITY
   +0x070 ServiceCount     : Uint4B
   +0x074 DispatchCount    : Uint4B
   +0x078 Rsvd1            : Uint8B
   +0x080 TrapFrame        : Ptr64 _KTRAP_FRAME
   +0x088 Reserved         : Ptr64 Void
   +0x090 DispatchCode     : [4] Uint4B
```

Следует заметить, что при выполнении команды dt подструктуры (структуры внутри структур) по умолчанию не показываются. Для выполнения рекурсии подструктур нужно воспользоваться ключом -r. Например, воспользоваться этим ключом для вывода объекта прерывания ядра с показом формата структуры _LIST_ENTRY, хранящейся в поле InterruptListEntry:

```
lkd> dt nt!_kinterrupt -r
nt!_KINTERRUPT
   +0x000 Type               : Int2B
   +0x002 Size               : Int2B
   +0x008 InterruptListEntry : _LIST_ENTRY
      +0x000 Flink           : Ptr64 _LIST_ENTRY
         +0x000 Flink        : Ptr64 _LIST_ENTRY
         +0x008 Blink        : Ptr64 _LIST_ENTRY
      +0x008 Blink           : Ptr64 _LIST_ENTRY
         +0x000 Flink        : Ptr64 _LIST_ENTRY
         +0x008 Blink        : Ptr64 _LIST_ENTRY
```

В файле справки Debugging Tools for Windows также объясняется, как настраиваются и используются отладчики ядра. Дополнительные подробности использования отладчиков ядра, предназначенные непосредственно для создателей драйверов устройств, могут быть найдены в документации по набору Windows Driver Kit.

Инструментальное средство LiveKd

LiveKd является свободно распространяемым средством, предлагаемым на сайте Sysinternals. Это средство позволяет использовать только что рассмотренные стандартные отладчики ядра компании Microsoft для исследования запущенной системы без загрузки этой системы в режиме отладки. Этот подход может пригодиться, когда причины возникновения проблем нужно установить на машине, не запущенной в режиме отладки, — иногда встречаются такие проблемы, которые очень трудно воспроизвести, а перезагрузка с включенным режимом отладки может не показать наличия какой-либо ошибки.

Средство LiveKd запускается точно так же, как WinDbg или Kd, и передает выбранному вами отладчику любые ключи, указанные в командной строке. По умолчанию LiveKd запускает отладчик ядра, работающий в окне командной строки — Kd. Чтобы это средство запустило WinDbg, нужно указать ключ -w. Для вывода файлов справки, касающихся ключей LiveKd, нужно указать ключ -?.

LiveKd представляет отладчику смоделированный файл аварийного дампа, поэтому в LiveKd можно выполнять любые операции, поддерживаемые в отношении аварийного дампа. Поскольку при моделировании аварийного дампа

средство LiveKd полагается на физическую память, отладчик ядра может попасть в такую ситуацию, при которой структуры данных находятся в центре области, изменяемой системой, и теряют свою согласованность. При каждом запуске отладчика он начинает со свежего представления состояния системы. Если нужно обновить снимок состояния (snapshot), выйдите из отладчика (с помощью команды q), и LiveKd выведет запрос на его повторный запуск. Если отладчик входит в цикл вывода на печать, нажмите сочетание клавиш Ctrl+C для прерывания вывода и выхода из отладчика. Если отладчик завис, нажмите сочетание клавиш Ctrl+Break, чтобы остановить процесс отладки. LiveKd выведет запрос на повторный запуск отладчика.

Windows Software Development Kit

Набор инструментальных средств Windows Software Development Kit (SDK) доступен в виде части программы подписки MSDN или же может быть свободно загружен с сайта msdn.microsoft.com. Кроме средств отладки Debugging Tools, он содержит документацию, заголовочные файлы языка C и библиотеки, необходимые для компиляции и компоновки Windows-приложений. (Хотя Microsoft Visual C++ поставляется с копией этих заголовочных файлов, версии, содержащиеся в Windows SDK, всегда соответствуют самым свежим версиям операционных систем Windows, а версии, поставляемые с Visual C++, могут быть более старыми, актуальными на время выхода Visual C++.) С точки зрения внутреннего устройства, Windows SDK интересны заголовочные файлы Windows API (\Program Files\Microsoft SDKs\Windows\v7.0A\Include). Некоторые из инструментальных средств этого набора также поставляются в виде экземпляров исходного кода, как в Windows SDK, так и в библиотеке MSDN Library.

Windows Driver Kit

Набор инструментальных средств Windows Driver Kit (WDK) также доступен по программе подписки MSDN и точно так же, как и Windows SDK, его можно свободно скачать с сайта. Документация по Windows Driver Kit включена в библиотеку MSDN.

Хотя набор WDK предназначен для разработчиков драйверов устройств, он является обширным источником информации о внутреннем устройстве Windows. А в документации по WDK содержится полное описание всех Windows-функций поддержки ядра и механизмов, используемых драйверами устройств как в виде учебного пособия, так и в виде справочника.

В WDK, помимо документации, содержатся заголовочные файлы (в частности, ntddk.h, ntifs.h и wdm.h), определяющие ключевую внутреннюю структуру данных и константы, а также интерфейсы ко многим внутренним системным подпрограммам. Эти файлы пригодятся при исследовании внутренних структур данных Windows, используемых при отладке ядра, поскольку, несмотря на то что общая схема и содержимое этих структур показаны в данной книге, подробные описания вплоть до каждого поля (например, размера и типа данных) в ней не даны. А в WDK дается полное описание таких структур (например, заголовков для диспетчера объектов, блоков ожидания, событий, мутантов, семафоров и т. д.).

При желании углубиться в изучение системы ввода-вывода и модели драйверов, выходя за рамки предлагаемого в данной книге, читайте WDK-документацию (особенно Руководство по устройству архитектуры драйверов, работающих в режиме ядра — Kernel-Mode Driver Architecture Design Guide, и Справочные руководства). Также могут пригодиться книги Уолтера Они (Walter Oney) «Использование Microsoft Windows Driver Model», второе издание (Питер, 2007) и Пенни Орвика (Penny Orwick) и Гая Смита (Guy Smith) «Windows Driver Foundation. Разработка драйверов» (BHV, 2008).

Инструментальные средства Sysinternals

Во многих экспериментах, рассматриваемых в данной книге, используется свободно распространяемый инструментарий, который можно загрузить с сайта Sysinternals. Большинство представленных там инструментальных средств создано соавтором этой книги Марком Руссиновичем (Mark Russinovich). Наиболее популярными средствами можно назвать Process Explorer и Process Monitor. Следует отметить, что многие из этих средств требуют установку и выполнение драйверов устройств, работающих в режиме ядра, а это, в свою очередь, требует повышенных прав администратора, но могут выполняться с ограниченными функциональными возможностями и с ограниченным выводом информации и при входе в систему с использованием стандартной (непривилегированной) учетной записи пользователя.

Поскольку инструментарий Sysinternals часто обновляется, лучше все-таки будет убедиться, что вы используете самую свежую версию. Для получения уведомлений об обновлении средств можно обратиться на блог сайта Sysinternals (у которого имеется RSS-канал).

Описание всех средств, порядка их использования, а также примеры решаемых с их помощью задач даны в книге Марка Руссиновича (Mark Russinovich) и Аарона Маргосиса (Aaron Margosis) «Windows Sysinternals Administrator's Reference» (Microsoft Press, 2011).

Для вопросов и обсуждений, касающихся этих средств, можно воспользоваться форумами Sysinternals.

Заключение

В данной главе были представлены ключевые технические понятия и термины Windows, используемые во всем остальном тексте книги. Вы также получили начальное представление о многих полезных инструментальных средствах, доступных для проникновения в глубь внутреннего устройства Windows. Теперь мы готовы приступить к исследованиям внутреннего устройства системы, начиная с общего вида системной архитектуры и ее ключевых компонентов.

Глава 2. Архитектура системы

После изучения терминов, понятий и инструментальных средств можно приступить к исследованию внутренних задач конструирования и структуры операционной системы Microsoft Windows. В данной главе рассматривается общая архитектура системы — основные компоненты, порядок их взаимодействия и контекст, в котором они работают. Чтобы заложить основы понимания внутреннего устройства Windows, сначала рассмотрим требования и цели, определяющие очертания исходной конструкции и спецификации системы.

Требования и цели разработки

В далеком 1989 году спецификация Windows NT определялась с учетом следующих требований:

- Создать по-настоящему передовую, 32-разрядную операционную систему, работающую с виртуальной памятью и допускающую повторный вход.
- Обеспечить возможность работы на разных аппаратных архитектурах и платформах.
- Обеспечить возможность работы и масштабирования на симметричных мультипроцессорных системах.
- Обеспечить возможность работы в качестве распределенной вычислительной платформы, как в роли сетевого клиента, так и в роли сервера.
- Обеспечить возможность запуска большинства существующих 16-разрядных приложений MS-DOS и Microsoft Windows 3.1.
- Обеспечить выполнение правительственных требований о совместимости со стандартом POSIX 1003.1.
- Обеспечить выполнение требований правительства и промышленности, касающихся безопасности операционных систем.
- Обеспечить адаптируемость к всемирному рынку за счет поддержки Unicode.

Чтобы заложить основы для принятия тысяч решений по созданию системы, отвечающей вышеперечисленным требованиям, команда разработчиков в самом начале работы над проектом решила реализовать следующие замыслы:

- **Расширяемость.** Код должен создаваться с учетом удобства его наращивания и изменения в соответствии с изменениями требований рынка.
- **Переносимость.** Система должна работать на разных аппаратных архитектурах, и, в соответствии с требованиями рынка, должна относительно легко переноситься на их новые образцы.
- **Надежность и отказоустойчивость.** Система должна защищать саму себя как от внешних угроз, так и от внутренних сбоев. Приложения не должны иметь возможность нанесения вреда операционной системе или другим приложениям.
- **Совместимость.** Хотя Windows NT должна была стать новым шагом по сравнению с существовавшей в то время технологией, ее пользовательский интерфейс и API должны быть совместимы с прежними версиями Windows

и с MS-DOS. У нее также должна быть возможность взаимодействия с другими системами, такими как UNIX, OS/2 и NetWare.

- **Производительность.** С учетом ограничений, накладываемых другими проектировочными замыслами, система должна проявлять максимально возможное быстродействие и реакцию на каждой аппаратной платформе.

По мере исследования подробностей внутренней структуры и работы Windows, вы увидите, как эти исходные замыслы проектирования и рыночные требования были успешно сплетены в конструкцию системы. Но прежде чем приступить к исследованиям, нужно изучить общую модель проектирования Windows и сравнить ее с моделями других современных операционных систем.

Модель операционной системы

В большинстве многопользовательских операционных систем приложения отделены от самой операционной системы. Код ядра операционной системы запускается в привилегированном режиме работы процессора (в данной книге он называется режимом ядра), имея доступ к системным данным и к оборудованию, а код приложения запускается в непривилегированном режиме работы процессора (пользовательский режим), с ограниченным набором доступных интерфейсов, ограниченным доступом к системным данным и без прямого доступа к оборудованию. Когда программа, запущенная в пользовательском режиме, вызывает системную службу, в процессоре выполняется специальная инструкция, переключающая вызывающий поток в режим ядра. Когда системная служба завершит свою работу, операционная система переключит контекст потока назад, в пользовательский режим, и позволит вызывающей программе продолжить свое выполнение. Windows похожа на большинство UNIX-систем тем, что она является монолитной операционной системой — в том смысле, что основная часть кода операционной системы и драйверов устройств совместно используют одно и то же защищенное пространство памяти, используемое в режиме ядра. Это означает, что любой компонент операционной системы или драйвер устройства потенциально может разрушить данные, используемые другими компонентами операционной системы. Но в Windows реализованы такие механизмы защиты ядра, как PatchGuard и Kernel Mode Code Signing (см. главу 3), помогающие ликвидировать и предотвратить проблемы, связанные с общим адресным пространством, используемым при работе в режиме ядра.

Разумеется, все эти компоненты операционной системы полностью защищены от неправильно работающих приложений, поскольку у этих приложений нет прямого доступа к коду и данным, находящимся в привилегированной части операционной системы (хотя они могут вызвать другие службы ядра). Эта защита является одной из причин наличия у Windows репутации отказоустойчивой и стабильной операционной системы, как в качестве сервера приложений, так и в качестве платформы для рабочей станции. Наряду с этим быстродействием Windows обладает такими службами ядра операционной системы, как управление виртуальной памятью, файловый ввод-вывод, сеть и совместное использование файлов и принтеров.

Компоненты Windows, работающие в режиме ядра, являются также воплощением принципов объектно-ориентированного проектирования. К примеру, они

вообще не проникают в чужие структуры данных для доступа к информации, поддерживаемой отдельными компонентами. Вместо этого для передачи параметров и доступа к структурам данных и (или) для их изменения используются формальные интерфейсы.

Несмотря на повсеместное использование объектов для представления общих системных ресурсов, Windows, в строгом смысле слова, объектно-ориентированной системой не является. Основная часть кода операционной системы написана на языке C для решения задач переносимости. Этот язык программирования не поддерживает напрямую такие объектно-ориентированные конструкции, как динамическое связывание типов данных, полиморфные функции или наследования классов. Поэтому имеющаяся в Windows реализация объектов на основе языка C заимствует свойства, присущие объектно-ориентированным языкам, но не зависит от них.

Краткий обзор архитектуры

После краткого обзора исходных замыслов проектирования и компоновки Windows рассмотрим основные компоненты системы, составляющие ее архитектуру. Упрощенная версия архитектуры показана на рис. 2.1. Нужно иметь в виду, что эта схема носит общий характер и не отображает все компоненты. (Например, на ней не показаны сетевые компоненты и иерархия различных типов драйверов устройств.)

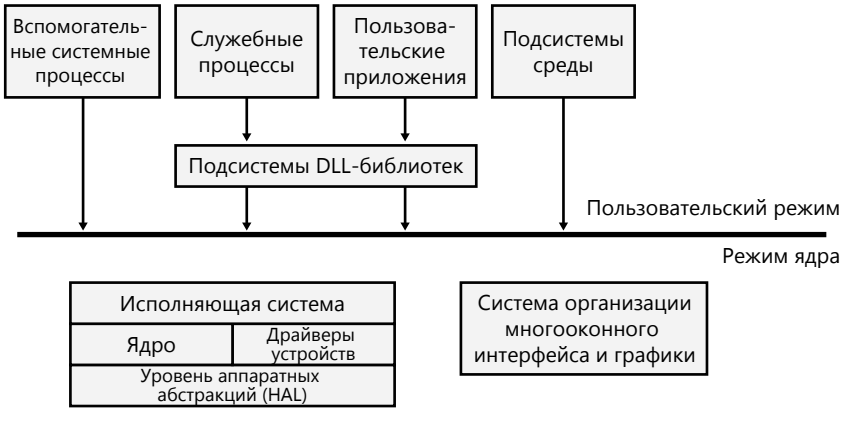


Рис. 2.1. Упрощенное представление архитектуры Windows

В первую очередь на рис. 2.1 нужно обратить внимание на прямую линию, разделяющую части операционной системы Windows, работающие в пользовательском режиме и в режиме ядра. Прямоугольники, находящиеся выше линии, представляют процессы, идущие в пользовательском режиме, а компоненты, показанные ниже линии, представляют системные службы, работающие в режиме ядра. Как уже говорилось в главе 1, потоки пользовательского режима выполняются в защищенном адресном пространстве процесса (когда они выполняются в режиме ядра, у них есть доступ к системному пространству). Таким образом, у вспомогательных системных процессов, у процессов служб, у пользовательских

приложений и у подсистем среды окружения, — у всех есть свое собственное закрытое адресное пространство.

Четырем основным процессам пользовательского режима можно дать следующие описания:

- Фиксированные (или реализованные на аппаратном уровне) *вспомогательные системные процессы*, такие как процесс входа в систему и администратор сеансов — Session Manager, которые не входят в службы Windows (они не запускаются диспетчером управления службами).
- *Служебные процессы*, реализующие такие службы Windows, как Диспетчер задач (Task Scheduler) и спулер печати (Print Spooler). Как правило, от служб требуется, чтобы они работали независимо от входов пользователей в систему. Многие серверные приложения Windows, такие как Microsoft SQL Server и Microsoft Exchange Server, также включают компоненты, работающие как службы.
- *Пользовательские приложения*, которые могут относиться к одному из следующих типов: для 32- или 64-разрядной версии Windows, для 16-разрядной версии Windows 3.1, для 16-разрядной версии MS-DOS или для 32- или 64-разрядной версии POSIX. Следует учесть, что 16-разрядные приложения могут запускаться только на 32-разрядной версии Windows.
- *Серверные процессы подсистемы окружения*, которые реализуют часть поддержки среды операционной системы или специализированную часть, представляемую пользователю и программисту. Изначально Windows NT поставляется тремя подсистемами среды: Windows, POSIX и OS/2. Но подсистемы POSIX и OS/2 последний раз поставлялись с Windows 2000. Выпуски клиентской версии Windows Ultimate и Enterprise, а также все серверные версии включают поддержку для усовершенствованной подсистемы POSIX, которая называется подсистемой для приложений на основе Unix (Unix-based Applications, SUA).

Обратите внимание на прямоугольник «Подсистемы DLL-библиотек», который на рис. 2.1 находится под прямоугольниками «Служебные процессы» и «Пользовательские приложения». При выполнении под управлением Windows пользовательские приложения не вызывают имеющиеся в операционной системе Windows службы напрямую, а проходят через одну или несколько подсистем динамически подключаемых библиотек (dynamic-link libraries, DLL). Подсистемы DLL-библиотек предназначены для перевода документированной функции в соответствующий внутренний (и зачастую недокументированный) вызов системной службы. Этот перевод может включать в себя (или не включать) отправку сообщения процессу подсистемы среды, обслуживающему пользовательское приложение.

В Windows входят следующие компоненты, работающие в режиме ядра:

- *Исполняющая система* Windows содержит основные службы операционной системы, такие как управление памятью, управление процессами и потоками, безопасность, ввод-вывод, сеть и связь между процессами.
- *Ядро* Windows состоит из низкоуровневых функций операционной системы, таких как диспетчеризация потоков, диспетчеризация прерываний и исключений и мультипроцессорная синхронизация. Оно также предоставляет набор подпрограмм и базовых объектов, используемых остальной исполняющей системой для реализации высокоуровневых конструктивных элементов.

- К *драйверам устройств* относятся как аппаратные драйверы устройств, которые переводят вызовы функций ввода-вывода в запросы ввода-вывода конкретного аппаратного устройства, так и неаппаратные драйверы устройств, такие как драйверы файловой системы и сети.
- *Уровень аппаратных абстракций* (hardware abstraction layer, HAL), являющийся уровнем кода, который изолирует ядро, драйверы устройств и остальную исполняющую систему Windows от аппаратных различий конкретных платформ (таких как различия между материнскими платами).
- *Система организации многооконного интерфейса и графики*, реализующая функции графического пользовательского интерфейса (graphical user interface, GUI), более известные как имеющиеся в Windows USER- и GDI-функции, предназначенные для работы с окнами, элементами управления пользовательского интерфейса и графикой.

В табл. 2.1 перечислены имена файлов основных компонентов операционной системы Windows. (Вам нужно знать эти имена файлов, потому что на некоторые системные файлы мы будем ссылаться по именам.) Каждый из этих компонентов будет более подробно рассмотрен либо в этой, либо в одной из следующих глав.

Таблица 2.1. Основные системные файлы Windows

Имя файла	Компоненты
Ntoskrnl.exe	Исполняющая система и ядро
Ntkrnlpa.exe (только в 32-разрядных системах)	Исполняющая система и ядро с поддержкой расширения физического адреса — Physical Address Extension (PAE), позволяющего 32-разрядным системам осуществлять адресацию вплоть до 64 Гб физической памяти и помечать память как не содержащую исполняемый код
Hal.dll	Уровень аппаратных абстракций
Win32k.sys	Часть подсистемы Windows, работающей в режиме ядра
Ntdll.dll	Внутренние вспомогательные функции и заглушки диспетчера системных служб к исполняющим функциям
Kernel32.dll, Advapi32.dll, User32.dll, Gdi32.dll	Основные Windows-подсистемы DLL-библиотек

Прежде чем приступить к подробному изучению этих компонентов системы, давайте рассмотрим некоторые основы конструкции ядра Windows. Начинем с того, как в Windows осуществляется переносимость, позволяющая этой операционной системе работать на нескольких аппаратных архитектурах.

Переносимость

Windows разрабатывалась для работы на разных аппаратных архитектурах. Исходный выпуск Windows NT поддерживал архитектуры x86 и MIPS. Вскоре после этого была добавлена поддержка процессора Alpha AXP компании Digital Equipment Corporation (она была куплена компанией Compaq, слившейся в последствии с компанией Hewlett-Packard). (Хотя процессор Alpha AXP был

64-разрядным, Windows NT запускалась в 32-разрядном режиме. В процессе разработки Windows 2000 на Alpha AXP была запущена собственная 64-разрядная версия, но она так и не была выпущена.) Поддержка четвертой процессорной архитектуры, Motorola PowerPC, была добавлена в Windows NT 3.51. Но из-за изменений рыночных потребностей к началу разработки Windows 2000 поддержка архитектур MIPS и PowerPC была прекращена. Чуть позже компания Compaq отозвала поддержку архитектуры Alpha AXP, и Windows 2000 стала поддерживать только архитектуру x86. В Windows XP и Windows Server 2003 была добавлена поддержка трех 64-разрядных процессорных семейств: Intel Itanium IA-64, AMD64 и 64-разрядных версий Intel Extension Technology (EM64T) для x86. (Они были совместимы с архитектурой AMD64, хотя и обладали некоторыми отличиями в поддерживаемых командах.) Последние два процессорных семейства получили название «64-разрядных расширенных систем», и в этой книге они будут рассматриваться как x64. Как Windows запускает 32-разрядные приложения на 64-разрядных версиях, будет показано в главе 3.

Переносимость Windows между аппаратными архитектурами и платформами достигается двумя основными способами:

- Windows имеет многоуровневую конструкцию, в которой нижний уровень системы предназначается для архитектуры конкретного процессора или конкретной платформы, и он выделен в отдельные модули, чтобы верхние уровни могли быть защищены от различий между архитектурами и между аппаратными платформами. Ключевыми компонентами, обеспечивающими переносимость операционной системы, являются ядро (которое содержится в файле Ntoskrnl.exe) и уровень аппаратных абстракций (или HAL, который содержится в библиотеке Hal.dll). Более подробно оба этих компонента еще будут рассмотрены в данной главе. Функции, зависящие от конкретной архитектуры (такие, как переключение контекста потока и диспетчеризация системных прерываний), реализованы в ядре. Функции, которые могут отличаться между системами в рамках одной архитектуры (например, при использовании разных материнских плат), реализованы в HAL. Единственным компонентом, имеющим существенный объем кода, предназначенного для конкретной архитектуры, является диспетчер памяти, но по сравнению со всей системой в целом этот объем незначителен.
- В подавляющем большинстве Windows написана на языке C, но встречаются и фрагменты, написанные на C++. Ассемблер используется только для тех частей операционной системы, которые должны напрямую взаимодействовать с системным оборудованием (например, обработчик системных прерываний) или для тех частей, работа которых сильно влияет на производительность системы (например, для переключения контекста). Код на языке ассемблера присутствует не только в ядре и на HAL-уровне, но также и в некоторых других основных местах операционной системы (например, в подпрограммах, реализующих инструкции взаимной блокировки, а также в модуле из вызова локальных процедур), в части подсистемы Windows, которая выполняется в режиме ядра, и даже в некоторых библиотеках пользовательского режима, например в коде запуска процесса в Ntdll.dll (в системной библиотеке, рассматриваемой далее в этой главе).

Симметричная мультипроцессорная обработка

Многозадачность представляет собой технологию операционной системы для совместного использования одного процессора несколькими потоками выполнения. Но когда у компьютера более одного процессора, он может выполнять несколько потоков одновременно. Таким образом, если многозадачная операционная система только кажется исполняющей одновременно несколько потоков, мультипроцессорная операционная система делает это на самом деле, выполняя по одному потоку на каждом своем процессоре.

Как говорилось в начале главы, одним из основных конструкторских замыслов для Windows была ее обязательная работа на мультипроцессорных компьютерных системах. Windows является *симметричной мультипроцессорной* (symmetric multiprocessing, SMP) операционной системой. В ней нет главного процессора — как операционная система, так и пользовательские потоки могут быть спланированы для работы на любом процессоре. Кроме этого, все процессоры совместно используют одно и то же адресное пространство. Эта модель отличается от *асимметричной мультипроцессорной обработки* (asymmetric multiprocessing, ASMP), при которой каждой операционной системе обычно выделяется один процессор для выполнения кода ядра операционной системы, а на других процессорах выполняется только пользовательский код. Различия между двумя мультипроцессорными моделями показаны на рис. 2.2.

Windows также поддерживает три современных типа мультипроцессорных систем: многоядерные, гиперпотоковость (Hyper-Threading) и с технологией доступа к неоднородной памяти — NUMA (non-uniform memory architecture). Они еще будут кратко упомянуты в следующих разделах. (Полное и подробное описание поддержки диспетчеризации этих систем будет дано в разделе, посвященном диспетчеризации потоков в главе 5 «Процессы, потоки и задания».)

Гиперпотоковость (Hyper-Threading) является технологией, представленной компанией Intel и предоставляющей два логических процессора для каждого физического ядра. У каждого логического процессора есть свое собственное состояние центрального процессора, но механизм исполнения команд и процессорная кэш-память у них общие. Это позволяет одному логическому центральному процессору успешно работать, в то время как другой логический центральный процессор остановлен (например, после промаха при обращении к кэш-памяти или после неправильно спрогнозированного условного перехода). Алгоритмы диспетчеризации усовершенствованы с учетом оптимального использования машин, допускающих гиперпотоковость, например путем диспетчеризации потоков на простаивающий физический процессор, а не на простаивающий логический процессор на физическом процессоре, где другой логический процессор занят работой. Более подробно диспетчеризация потоков рассмотрена в главе 5.

На NUMA-системах процессоры сгруппированы в небольшие блоки, называемые *узлами*. У каждого узла есть свои собственные процессоры и память, и он подключен к более крупной системе через кэш-когерентную объединяющую шину. Тем не менее Windows на NUMA-системе работает как SMP-система, в которой все процессоры имеют доступ ко всей памяти — при том, что обращение к локальной памяти узла осуществляется быстрее, чем к памяти, подключенной к другим узлам. Система пытается повысить производительность путем диспетчеризации потоков на процессах того же узла, на котором находится используемая

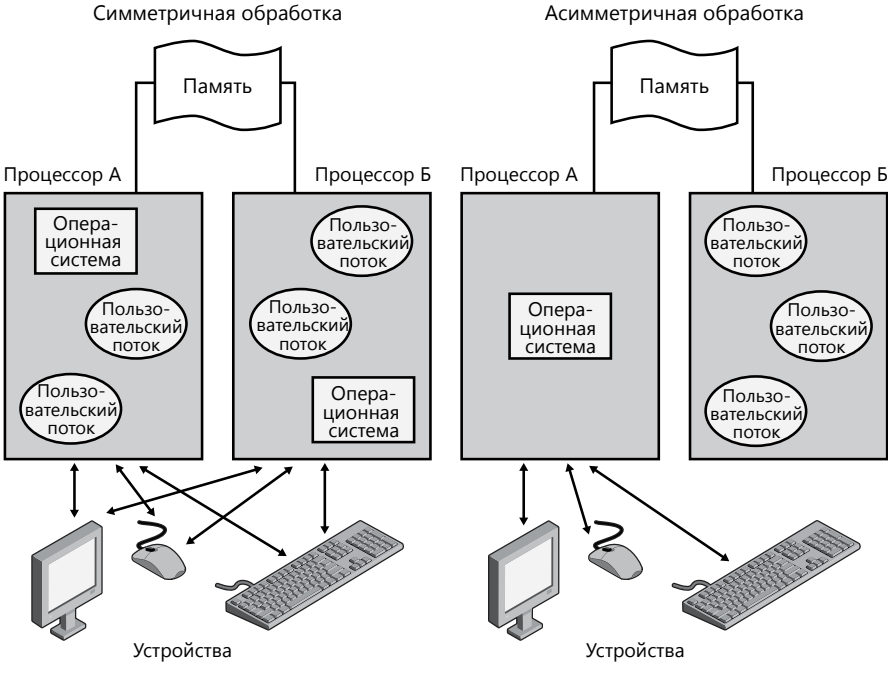


Рис. 2.2. Сравнение симметричной и асимметричной мультипроцессорной обработки

ими память. Она пытается удовлетворить запросы на выделение памяти в рамках узла, но при необходимости выделит память, подключенную и к другим узлам.

И конечно же, Windows также изначально поддерживает многоядерные системы, поскольку у этих систем есть настоящие физические ядра (они находятся на одном кристалле), исходный SMP-код в Windows считает их отдельными процессорами, за исключением некоторых задач учета использования ресурсов и идентификации (например, лицензирования), делающих различия между ядрами на одном и том же процессоре и ядрами на разных сокетах.

Изначально в Windows не предусматривался какой-либо лимит на количество процессоров, если не считать политик лицензирования, устанавливающих различия между разными версиями Windows. Но в целях удобства и эффективности Windows занимается отслеживанием процессоров (их общего числа, простоев, занятости и других таких же подробностей) в битовой маске (маской родственности, affinity mask), в которой количество бит соответствует исходной размерности данных машины (32- или 64-разрядной), что позволяет процессору манипулировать битами непосредственно в регистре. Из-за этого количество процессоров в системе Windows изначально ограничивалось исходной размерностью слова, поскольку маска родственности не могла быть увеличена произвольным образом. Чтобы обеспечить совместимость, а также поддержку более крупных процессорных систем, в Windows реализована конструкция более высокого порядка, называемая группой процессоров. Такая группа является набором процессоров, где все процессоры группы могут быть определены единой битовой маской родственности, и ядро операционной системы, как и приложения, могут

выбрать группу, к которой они обращаются при обновлении маски родственности. Совместимые с данной конструкцией приложения могут запросить количество поддерживаемых групп (ограничиваемое в настоящее время числом 4) а затем подсчитать битовую маску для каждой группы. При этом устаревшие приложения продолжают работать, видя только текущую группу. Информация о том, как именно Windows назначает процессоры для групп (что также относится и к NUMA), дается в главе 5.

Как уже упоминалось, фактическое число поддерживаемых лицензированных процессоров зависит от используемой версии Windows. (См. далее табл. 2.2.) Это количество хранится в файле лицензионной политики системы (\Windows\ServiceProfiles\NetworkService\AppData\Roaming\Microsoft\Software-ProtectionPlatform\tokens.dat) в виде значения политики под названием «Kernel-RegisteredProcessors». Следует иметь в виду, что фальсификация данных является нарушением лицензии на программное обеспечение, и изменение лицензионной политики с целью использования большего количества процессоров влечет за собой более серьезные последствия, чем простое изменение этого значения.

Масштабируемость

Одним из ключевых вопросов мультипроцессорных систем является масштабируемость. Для корректной работы на SMP-системе код операционной системы должен следовать строгим принципам и правилам. Борьба за ресурсы и решение других проблем, влияющих на производительность системы, в мультипроцессорных системах происходит сложнее, чем в однопроцессорных. Это должно быть учтено в конструкциях мультипроцессорных систем. В Windows имеется ряд функций, имеющих решающее значение для достижения ее успешной работы в качестве мультипроцессорной операционной системы:

- Возможность запуска кода операционной системы на любом доступном процессоре и одновременно на нескольких процессорах.
- Несколько потоков выполнения в одном процессе, каждый из которых может одновременно выполняться на разных процессорах.
- Тонкая синхронизация внутри ядра (с помощью спин-блокировок, спин-блокировок с очередью и пуш-блокировок, рассматриваемых в главе 3) наряду с драйверами устройств и служебными процессами, позволяющая большему количеству компонентов запускаться параллельно на нескольких процессорах.
- Программирование таких механизмов, как порты завершения ввода-вывода, способствующих эффективной реализации многопоточных серверных процессов, хорошо масштабируемых на мультипроцессорных системах.

Со временем масштабируемость ядра Windows улучшилась. Например, в Windows Server 2003 были представлены очереди планирования для каждого процессора, позволяющие принимать решения по диспетчеризации потоков параллельно на нескольких процессорах. В Windows 7 и Windows Server 2008 R2 была убрана глобальная блокировка в отношении диспетчеризации баз данных. Такому поэтапному улучшению детализации блокировки подверглись и другие области, например диспетчер памяти. Дополнительную информацию о мультипроцессорной синхронизации можно найти в главе 3.

Различия между клиентскими и серверными версиями

Windows поставляется как в клиентских, так и в серверных версиях. На момент написания данной книги существовало шесть клиентских версий Windows 7: Windows 7 Home Basic, Windows 7 Home Premium, Windows 7 Professional, Windows 7 Ultimate, Windows 7 Enterprise и Windows 7 Starter.

Существует семь различных серверных версий Windows Server 2008 R2: Windows Server 2008 R2 Foundation, Windows Server 2008 R2 Standard, Windows Server 2008 R2 Enterprise, Windows Server 2008 R2 Datacenter, Windows Web Server 2008 R2, Windows HPC Server 2008 R2 и Windows Server 2008 R2 for Itanium-Based Systems (выпуском Windows для процессора Intel Itanium).

Кроме этого существуют клиентские «N»-версии, не включающие в себя Windows Media Player. И наконец, версии Windows Server 2008 R2 Standard, Enterprise и Datacenter также включают выпуски «с Hyper-V», в которых присутствует Hyper-V. (Виртуализация Hyper-V рассматривается в главе 3.)

Все эти версии отличаются друг от друга следующими показателями:

- числом поддерживаемых процессоров (в понятиях сокетов, а не ядер или потоков);
- объемом поддерживаемой физической памяти (фактически, самый большой физический адрес, доступный для оперативной памяти);
- количеством поддерживаемых параллельных сетевых подключений (Например, в клиентской версии к файловым и принтерным службам допускается максимально 10 параллельных подключений.);
- поддержкой Media Center;
- поддержкой Multi-Touch, Aero и Диспетчера рабочего стола (Desktop Compositing);
- поддержкой таких свойств, как BitLocker, VHD Booting, AppLocker, Windows XP Compatibility Mode и более ста других значений настраиваемой политики лицензирования;
- многоуровневыми службами, поставляемыми с версиями Windows Server и не поставляемыми с клиентскими версиями (например, службами каталогов и кластеризации).

Различия в поддержке памяти и процессоров для Windows 7 и Windows Server 2008 R2 показаны в табл. 2.2. Подробная сравнительная таблица различных версий Windows Server 2008 R2 представлена на веб-сайте www.microsoft.com/windowsserver2008/en/us/r2-compare-specs.aspx.

Несмотря на то что операционная система Windows распространяется в виде нескольких клиентских и серверных пакетов поставки, все они используют один и тот же набор основных системных файлов, включая образ ядра, Ntoskrnl.exe (а в PAE-версии Ntkrnlpa.exe), HAL-библиотеки, драйверы устройств и базовые системные утилиты и DLL-библиотеки. Эти файлы идентичны для всех версий Windows 7 и Windows Server 2008 R2.

Откуда, при наличии такого разнообразия версий Windows с одинаковым образом ядра, система знает, какую именно версию загружать? Для этого делается запрос значений реестра ProductType и ProductSuite, находящихся в разделе HKLM\SYSTEM\CurrentControlSet\Control\ProductOptions. Значение ProductType используется для того, чтобы отличить клиентскую систему от серверной (любой разновидности). Эти значения загружаются в реестр на основе

Таблица 2.2. Различия между Windows 7 и Windows Server 2008 R2

	Количество поддерживаемых сокетов (32-разр. версия)	Объем поддерживаемой физической памяти (32-разр. версия), Гбайт	Количество поддерживаемых сокетов (64-разр. версия)	Объем поддерживаемой физической памяти (Itanium-версии), Гбайт	Объем поддерживаемой физической памяти (x64-версии), Гбайт
Windows 7 Starter 1	1	2	Нет	Нет	2
Windows 7 Home Basic	1	4	1	Нет	8
Windows 7 Home Premium	1	4	1	Нет	16
Windows 7 Professional	2	4	2	Нет	192
Windows 7 Enterprise	2	4	2	Нет	192
Windows 7 Ultimate	2	4	2	Нет	192
Windows Server 2008 R2 Foundation	Нет	Нет	1	Нет	8
Windows Web Server 2008 R2	Нет	Нет	4	Нет	32
Windows Server 2008 R2 Standard	Нет	Нет	4	Нет	32
Windows HPC Server 2008 R2	Нет	Нет	4	Нет	128
Windows Server 2008 R2 Enterprise	Нет	Нет	8	Нет	2048
Windows Server 2008 R2 Datacenter	Нет	Нет	64	Нет	2048
Windows Server 2008 R2 for Itanium-Based Systems	Нет	Нет	64	2048	Нет

рассмотренного ранее файла политики лицензирования. Допустимые значения перечислены в табл. 2.3. Это значение может быть запрошено из функции пользовательского режима `GetVersionEx` или из драйвера устройства с помощью вспомогательной функции режима ядра `RtlGetVersion`.

Таблица 2.3. Значения параметра ProductType, имеющегося в реестре

Версия Windows	Значение ProductType
Windows client	WinNT
Windows server (контроллер домена)	LanmanNT
Windows server (только сервер)	ServerNT

Другое значение реестра, ProductPolicy, содержит кэшированную копию данных, находящихся в файле tokens.dat, который устанавливает различия между версиями Windows и допускаемыми в них функциями.

Если пользовательским программам нужно определить, под какой версией Windows они работают, они могут вызвать Windows-функцию `VerifyVersionInfo` (см. документацию по SDK). Драйверы устройств могут вызвать функцию режима ядра `RtlVerifyVersionInfo` (см. документацию по WDK).

Но если основные файлы, по сути, одинаковы для клиентской и серверной версий, чем системы отличаются в работе? Вкратце, серверные системы по умолчанию оптимизированы под системную пропускную способность, позволяющую им выступать в роли высокопроизводительных серверов приложений, а клиентская версия (при наличии серверных возможностей) оптимизирована по времени отклика для интерактивного использования в качестве рабочего стола. Например, на основе типа продукта по-другому принимается ряд решений по распределению ресурсов в процессе загрузки системы. В частности, это касается размеров и количества областей памяти, выделяемых программе для динамически размещаемых структур данных (или пулов), количества внутренних рабочих потоков системы и размера кэш-памяти системных данных. Также серверная и клиентская версии отличаются друг от друга решениями политики времени выполнения, способом учета диспетчером памяти потребностей в системной памяти и в памяти процессов. Отличия между двумя семействами прослеживаются даже в некоторых деталях диспетчеризации потоков, составляющих их поведение по умолчанию (см. главу 5). Все существенные функциональные различия между двумя продуктами выделены в соответствующих главах данной книги. Если не сделано специальных оговорок, то все, описанное в данной книге, относится как к клиентским, так и к серверным версиям.

ЭКСПЕРИМЕНТ: ОПРЕДЕЛЕНИЕ ВОЗМОЖНОСТЕЙ, РАЗРЕШЕННЫХ ПОЛИТИКОЙ ЛИЦЕНЗИРОВАНИЯ

Как уже ранее упоминалось, Windows поддерживает более ста различных функций, которые могут быть разрешены посредством механизма лицензирования программного обеспечения. Соответствующие настройки политики определяют различия не только между клиентской и серверной установками, но также и отличие каждой версии (или идентификатора товарной позиции — stock-keeping unit, SKU) операционной системы, в частности это касается поддержки такого средства, как BitLocker (доступного на серверных версиях Windows, а также на клиентских версиях Windows Ultimate и Enterprise). Для отображения значений политики, определенной для вашей машины, можно

воспользоваться средством SlPolicy, доступным на веб-сайте Winsider Seminars & Solutions (www.winsiderss.com/tools/slpolicy.htm).

Настройки политики организованы по объектам, представляющим владельца модуля, к которому применяется политика. Запустив программу Slpolicy.exe с ключом –f, можно вывести список всех объектов, имеющихся в вашей системе:

```
C:\>SlPolicy.exe -f
SlPolicy v1.05 - Show Software Licensing Policies
Copyright (C) 2008-2011 Winsider Seminars & Solutions Inc.
www.winsiderss.com
Software Licensing Facilities:
Kernel
Licensing and Activation
Core
DWM
SMB
IIS
.
.
.
```

Чтобы вывести значение политики в отношении любого объекта, можно после ключа добавить его имя. Например, чтобы просмотреть ограничения, касающиеся центральных процессоров, доступной памяти нужно указать объект ядра — Kernel. Для машины с запущенной системой Windows 7 Ultimate можно ожидать следующий вывод:

```
C:\>SlPolicy.exe -f Kernel
SlPolicy v1.05 - Show Software Licensing Policies
Copyright (C) 2008-2011 Winsider Seminars & Solutions Inc.
www.winsiderss.com
Kernel
------
Processor Limit: 2
Maximum Memory Allowed (x86): 4096
Maximum Memory Allowed (x64): 196608
Maximum Memory Allowed (IA64): 196608
Maximum Physical Page: 4096
Addition of Physical Memory Allowed: No
Addition of Physical Memory Allowed, if virtualized: Yes
Product Information: 1
Dynamic Partitioning Supported: No
Virtual Dynamic Partitioning Supported: No
Memory Mirroring Supported: No
Native VHD Boot Supported: Yes
Bad Memory List Persistance Supported: No
Number of MUI Languages Allowed: 1000
List of Allowed Languages: EMPTY
List of Disallowed Languages: EMPTY
MUI Language SKU:
Expiration Date: 0
```

Отладочная сборка

Существует версия Windows, называемая отладочной сборкой (checked build), которая доступна только подписчикам MSDN Operating Systems. Это перекомпилированный исходный код Windows с выставленным флажком времени компиляции DBG (включает условный код отладки и трассировки). Кроме того, чтобы было проще понять машинный код, не производится последующая обработка двоичных кодов Windows, оптимизирующая расположение кода для быстрого выполнения. (См. раздел «Debugging Performance-Optimized Code» в файле справки Debugging Tools for Windows.)

В первую очередь отладочная сборка предназначена для помощи разработчикам драйверов устройств, потому что в ней выполняются более строгие проверки на наличие ошибок в функциях режима ядра, вызываемых драйверами устройств или другим системным кодом. Например, если драйвер (или какой-нибудь другой фрагмент кода, выполняемого в режиме ядра) осуществляет неверный вызов системной функции, которая ведет проверку аргументов (например, запрос спинблокировки на неправильном уровне прерывания), система при обнаружении проблемы останавливает выполнение, не допуская разрушения какой-нибудь структуры данных и возможной в последующем аварии системы.

ЭКСПЕРИМЕНТ: ОПРЕДЕЛЕНИЕ ФАКТА ЗАПУСКА ОТЛАДОЧНОЙ СБОРКИ

Чтобы вывести на экран информацию о том, какая именно сборка запущена, отладочная или поступающая в продажу (которая называется свободной), встроенного средства не существует. Но эта информация доступна через свойство «Debug» класса `Win32_OperatingSystem` инструментария управления Windows Management Instrumentation (WMI). Значение этого свойства выводится с помощью следующего примера сценария Microsoft Visual Basic:

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

Чтобы увидеть его в работе, наберите предыдущий код и сохраните его в файле.

```
C:\>cscript osversion.vbs
```

При запуске сценария на экран будет выведена следующая информация:

```
Сервер сценариев Windows (Microsoft R) версия 5.8
© Корпорация Майкрософт (Microsoft Corp.), 1996-2001. Все права защищены.

Заголовок: Microsoft Windows 7 Ultimate
Отладка: Ложь
Версия: 6.1.7600
```

Эта система запущена не из отладочной сборки, поскольку показанный здесь флажок отладки не установлен, то есть имеет значение `False` (Ложь). ■

Основная часть дополнительного кода в исполняемых файлах отладочной версии является результатом использования макроса ASSERT и (или) макроса NT_ASSERT, которые определены в заголовочном файле WDK Wdm.h и описаны в WDK-документации. Макрос проверяет условие (например, приемлемость структуры данных или параметра), и если выражение вычисляется в FALSE, макрос вызывает функцию RtlAssert, работающую в режиме ядра, которая вызывает функцию DbgPrintEx для отправки текста отладочного сообщения в предназначенный для этого сообщения буфер. Если подключен отладчик ядра, это сообщение выводится автоматически вместе с вопросом пользователю о том, что делать в связи с сообщением об отказе (установить контрольную точку, проигнорировать, завершить процесс или завершить поток). Если система не была запущена с отладчиком ядра (с использованием параметра отладки в базе данных конфигурации загрузки — Boot Configuration Database, BCD) и отладчик ядра не подключен, неудачное выполнение теста утверждения — ASSERT приведет к ошибке проверки системы. Перечень проверок ASSERT, проводимых некоторыми подпрограммами поддержки ядра, приводится в разделе «Checked Build ASSERTs» WDK-документации.

Отладочная сборка может также пригодиться системным администраторам своей дополнительной подробной информационной трассировкой, которая может быть включена для некоторых компонентов. (Подробные инструкции даны в статье базы знаний Microsoft «HOWTO: Enable Verbose Debug Tracing in Various Drivers and Subsystems».) Этот информационный вывод отправляется во внутренний буфер отладочных сообщений с использованием ранее упомянутой функции DbgPrintEx. Для просмотра отладочных сообщений можно либо подключить к целевой системе отладчик ядра (что требует загрузки целевой системы в отладочном режиме), воспользовавшись после этого командой !dbgprint при осуществлении отладки локального ядра, либо воспользоваться средством Dbgview.exe из набора Sysinternals (www.microsoft.com/technet/sysinternals).

Чтобы воспользоваться отладочной версией операционной системы необязательно устанавливать всю отладочную сборку. Достаточно в обычную поставляемую установку скопировать отладочную версию образа ядра (Ntoskrnl.exe) и соответствующую HAL-библиотеку (Hal.dll). Преимущество такого подхода заключается в том, что драйверы устройств и другие фрагменты кода ядра получают строгий контроль, присущий отладочной сборке без необходимости запуска работающих медленнее отладочных версий всех компонентов системы. Подробные инструкции приведены в разделе «Installing Just the Checked Operating System and HAL» WDK-документации.

И наконец, отладочная сборка может также пригодиться для тестирования кода, выполняемого в пользовательском режиме, что обусловлено только лишь разницей в синхронизации системы. (Причина в том, что в ядре проводятся дополнительные проверки и компоненты скомпилированы без оптимизации.) Зачастую ошибки многопоточной синхронизации связаны с особыми условиями ее организации. При запуске тестов на системе, где работает отладочная сборка (или, как минимум, отладочная версия ядра и соответствующая HAL-библиотека), сам факт другой синхронизации всей системы может выявить скрытые ошибки синхронизации, которые не проявляют себя на системах их обычных комплектов поставки.

Основные компоненты системы

После знакомства с высокоуровневой архитектурой Windows давайте углубимся в ее внутреннюю структуру и посмотрим, какую роль играет каждый основной компонент операционной системы. На рис. 2.3 показана более подробная

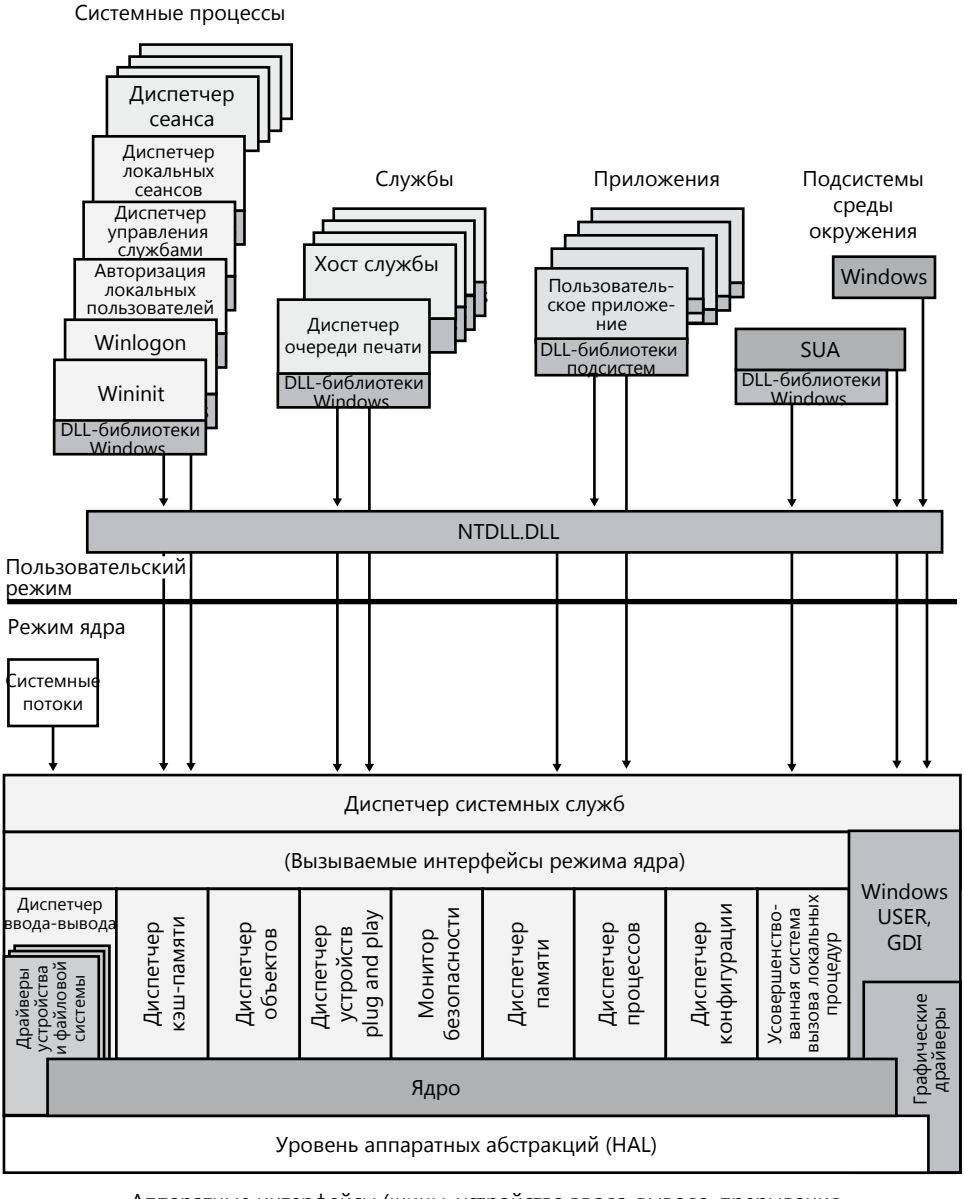


Рис. 2.3. Архитектура Windows

и полная схема архитектуры ядра и компонентов Windows, показанных ранее в данной главе на рис. 2.1. Следует учесть, что на этой схеме показаны еще не все компоненты (в частности, отсутствуют сетевые компоненты, рассматриваемые в главе 7 «Сеть»).

Каждый основной элемент данной схемы подробно рассмотрен в следующих разделах. В главе 3 объясняются первичные механизмы управления, используемые системой (например, диспетчер объектов, прерывания и т. д.). В главе 4 рассматриваются подробности таких механизмов управления, как реестр, служебные процессы и инструментарий управления Windows Management Instrumentation. В остальных главах еще более подробно исследуется внутренняя структура и работа таких ключевых областей, как процессы и потоки, диспетчер памяти, система безопасности, диспетчер ввода-вывода, управление хранением данных, диспетчер кэш-памяти, файловая система Windows (NTFS) и сеть.

Подсистемы среды окружения и DLL-библиотеки подсистем

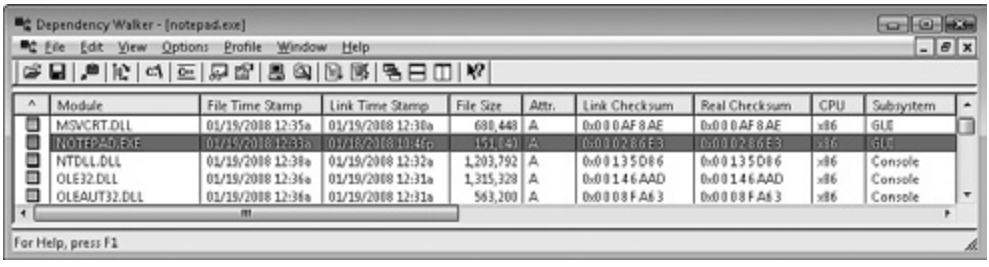
Роль подсистемы среды окружения заключается в предоставлении прикладным программам того или иного поднабора базовых служб исполняющей системы Windows. Каждая подсистема может предоставить доступ к различным поднаборам служб, присущих Windows. Это означает, что из приложений, построенных на использовании одной подсистемы, могут выполняться некие действия, которые не могут выполняться приложением, построенным на использовании другой подсистемы. Например, приложение Windows не может использовать SUA-функцию `fork`.

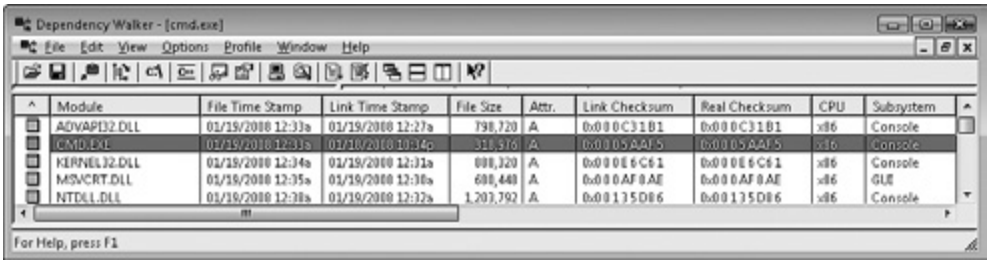
Каждый исполняемый образ (.exe) привязан к одной и только к одной подсистеме. При запуске образа код создания процесса исследует код типа подсистемы в заголовке образа, чтобы уведомить соответствующую подсистему о новом процессе. В Microsoft Visual C++ этот код типа указывается с помощью спецификатора `/SUBSYSTEM` команды `link`.

Как уже ранее упоминалось, пользовательские приложения не вызывают напрямую системные службы Windows. Вместо этого ими используется одна или несколько DLL-библиотек подсистемы. Эти библиотеки экспортируют документированный интерфейс, который может быть использован программами, связанными с данной подсистемой. Например, API-функции Windows реализованы в DLL-библиотеках подсистемы Windows, таких, как Kernel32.dll, Advapi32.dll, User32.dll и Gdi32.dll. А API-функции SUA реализованы в DLL-библиотеке подсистемы SUA (Psxdll.dll).

ЭКСПЕРИМЕНТ: ПРОСМОТР ТИПА ПОДСИСТЕМЫ ОБРАЗА

Тип подсистемы образа можно просмотреть с помощью средства Dependency Walker (Depends.exe) (доступно на сайте www.dependencywalker.com). Например, обратите внимание на типы образа для двух различных Windows-образов, Notepad.exe (простой текстовый редактор) и Cmd.exe (командная строка Windows):





Здесь показано, что Notepad (Блокнот) является GUI-программой, а Cmd является консольной (console), или текстовой программой. И хотя это означает, что существует две разные подсистемы для GUI-программ и текстовых программ, на самом деле есть только одна подсистема Windows, и GUI-программы могут иметь консоли, а консольные программы могут отображать GUI-элементы. ■

Когда приложение вызывает функцию, находящуюся в DLL-библиотеке подсистемы, может реализоваться одно из трех обстоятельств:

- Функция полностью реализована в режиме пользователя в DLL-библиотеке подсистемы. Иными словами, процессу подсистемы среды никакое сообщение не отправляется и никакие службы исполняющей системы Windows не вызываются. Функция выполняется в пользовательском режиме, а результаты возвращаются вызвавшему ее коду. Примерами таких функций могут послужить:
  - `GetCurrentProcess` (всегда возвращающая `-1`, значение, определяемое во всех, связанных с процессами функциях для ссылки на текущий процесс);
  - `GetCurrentProcessId` (ID процесса для запущенного процесса не изменяется, поэтому этот ID извлекается из ячейки кэш-памяти, исключая тем самым необходимость обращения к ядру).
- Функция требует один или несколько вызовов исполняющей системы Windows. Например, Windows-функции `ReadFile` и `WriteFile` включают соответственно вызовы базовых внутренних (и недокументированных) системных служб ввода-вывода Windows `NtReadFile` и `NtWriteFile`.
- Функция требует проведения в процессе подсистемы среды окружения некоторой работы. (Процессы подсистем среды окружения, запущенные в пользовательском режиме, отвечают за обслуживание состояния клиентских приложений, находящихся под их управлением.) В этом случае к подсистеме среды окружения делается клиент-серверный запрос посредством отправки подсистеме

сообщения для выполнения той или иной операции. Затем DLL-библиотека подсистемы, прежде чем вернуть управление вызывающему коду, ждет ответа.

Некоторые функции могут представлять собой комбинацию из только что перечисленных второго и третьего вариантов. В качестве примеров можно привести Windows-функции `CreateProcess` и `CreateThread`.

Запуск подсистем

Подсистемы запускаются процессом Диспетчера сеанса — Session Manager (Smss.exe). Информация, касающаяся запуска подсистем, хранится в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\SubSystems. Значения, хранящиеся в этом разделе, показаны на рис. 2.4.

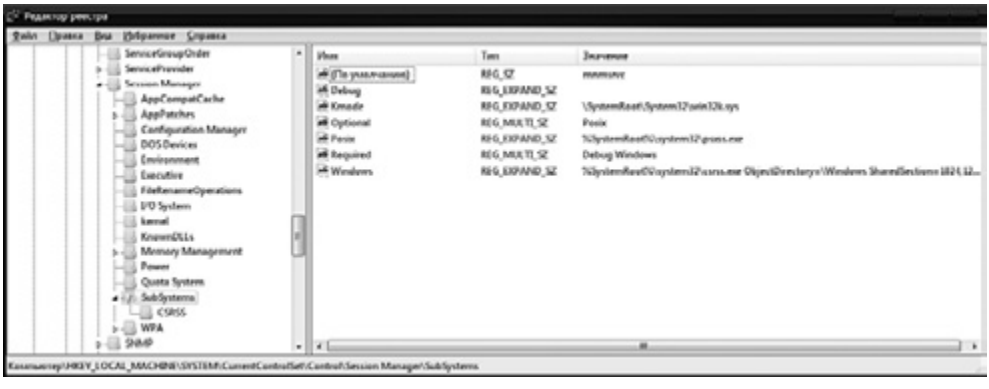


Рис. 2.4. Редактор реестра показывает информацию, касающуюся запуска подсистем

В значении `Required` перечисляются подсистемы, загружаемые при загрузке системы. Значение содержит две строки: `Windows` и `Debug`. Значение `Windows` содержит спецификацию файла подсистемы Windows, Csrss.exe (от Client/Server Run-Time Subsystem, клиент-серверная подсистема времени выполнения). Значение `Debug` оставлено пустым (поскольку оно используется для внутреннего тестирования) и поэтому не вызывает никаких действий. Значение `Optional` показывает, что по запросу будет запущена подсистема SUA. Значение реестра `Kmode` содержит имя файла той части подсистемы Windows, которая работает в режиме ядра, Win32k.sys (об этом мы поговорим чуть позже).

Давайте более пристально присмотримся к каждой из подсистем среды окружения.

Подсистема Windows

Хотя Windows была разработана для поддержки нескольких независимых подсистем среды окружения, с практической точки зрения наличие в каждой подсистеме всего кода для организации многооконного интерфейса и отображения ввода-вывода может привести к большой степени дублированности системных функций, что, в конечном счете, негативно скажется как на размере, так и на

производительности системы. Поскольку Windows была первичной подсистемой, разработчики Windows решили поместить эти базовые функции в ней и заставить другие подсистемы для отображения ввода-вывода вызывать подсистему Windows. Таким образом, подсистема SUA для отображения ввода-вывода вызывает службы в подсистеме Windows.

В результате такого конструкторского решения подсистема Windows является необходимым компонентом для любой Windows-системы, даже на серверных системах без зарегистрированных интерактивных пользователей. По этой причине процесс помечен как необходимый (стало быть, если по каким-то причинам происходит выход из этого процесса, система дает сбой).

Подсистема Windows состоит из следующих основных компонентов:

- Для каждого сеанса экземпляр процесса подсистемы среды (Csrss.exe) загружает три DLL-библиотеки (Basesrv.dll, Winsrv.dll и Csrsrv.dll), содержащие поддержку:
  - создания и удаления процессов и потоков;
  - частей, поддерживающих процессы 16-разрядной виртуальной DOS-машины (VDM) (только для 32-разрядной версии Windows);
  - Side-by-Side (SxS) сборок (Fusion) и манифестов;
  - других разнообразных функций, например `GetTempFile`, `DefineDosDevice`, `ExitWindowsEx` и нескольких естественных функций поддержки языка.
- Драйвер устройства режима ядра (Win32k.sys), включающий в себя:
  - диспетчер окон, который управляет выводом окон, осуществляет экранный вывод, получает ввод с клавиатуры, мыши и других устройств и передает приложениям пользовательские сообщения;
  - интерфейс графических устройств — Graphics Device Interface (GDI), представляющий собой библиотеку функций для устройств графического вывода. Он включает функции для рисования прямых линий, текста и фигур и для манипулирования графическими объектами;
  - оболочки для поддержки набора DirectX, реализуемого в другом драйвере ядра (Dxgkrnl.sys).
- Хост-процесс консоли — console host process (Conhost.exe), — предоставляющий поддержку для консольных (символьных) приложений.
- DLL-библиотеки подсистем (например, Kernel32.dll, Advapi32.dll, User32.dll и Gdi32.dll), которые превращают документированные функции Windows API в соответствующие и большей частью недокументированные вызовы системных служб режима ядра в Ntoskrnl.exe и Win32k.sys.
- Драйверы графических устройств для аппаратно-зависимых драйверов графических дисплеев, драйверов принтеров и драйверов видеомини-портов.

Для создания на дисплее таких элементов управления пользовательского интерфейса, как окна и кнопки, приложения вызывают стандартные USER-функции. Диспетчер окон передает соответствующие требования интерфейсу графических устройств GDI, а тот передает их драйверам графических устройств, где они имеют формат, соответствующий устройству отображения. Для завершения поддержки видеодисплея драйвер дисплея работает в паре с драйвером видеомини-порта.

ПРИМЕЧАНИЕ

В той части оптимизации работ, которая в Windows-архитектуре называется MinWin, DLL-библиотеки подсистем в настоящее время, как правило, состоят из специфических библиотек, реализующих API-наборы, которые затем компонуются вместе в DLL-библиотеку подсистемы и разрешаются с помощью специальной схемы перенаправления. Более подробная информация об этой оптимизации дана в главе 3, в разделе «Загрузчик образов».

GDI предоставляет набор стандартных двумерных функций, позволяющих приложениям обмениваться данными с графическими устройствами, ничего не зная об этих устройствах. GDI-функции являются посредниками между приложениями и такими графическими устройствами, как драйверы дисплеев и принтеров. GDI-интерфейс переводит запросы приложения на графический вывод и отправляет запросы драйверам графического дисплея. Он также предоставляет стандартный интерфейс для приложений для использования различных устройств графического вывода. Этот интерфейс позволяет коду приложения быть независимым от аппаратных устройств и их драйверов. GDI приспосабливает свои сообщения под возможности устройства, зачастую разделяя запрос на управляемые части. Например, некоторые устройства могут понимать направления рисования эллипса, в то время как другие требуют от GDI интерпретировать команду в виде серии пикселов, размещаемых по соответствующим координатам. (Дополнительная информация, касающаяся графики и архитектуры видеодрайверов, дана в Windows Driver Kit, в разделе «Design Guide» главы «Display (Adapters and Monitors)».)

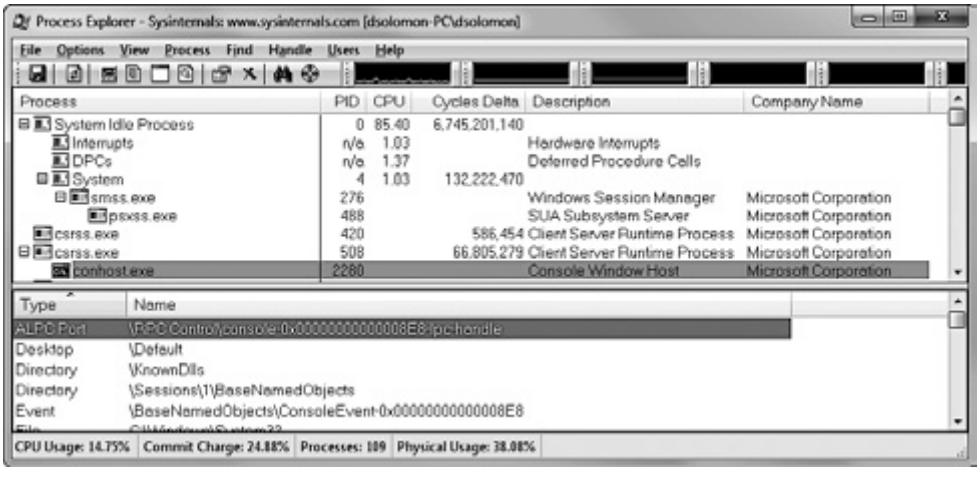
Поскольку основная часть подсистемы, в частности функции дисплейного ввода-вывода, работает в режиме ядра, сообщения процессу подсистемы Windows отправляются всего лишь несколькими Windows-функциями: создания и завершения процесса и потока, отображения буквы сетевого диска и создания временных файлов. В общем, работающее Windows-приложение не будет вызывать многочисленных переключений к процессу подсистемы Windows (или вообще не будет вызывать таких переключений).

ХОСТ ОКНА КОНСОЛИ

В исходной конструкции подсистемы Windows процесс подсистемы (Csrss.exe) отвечал за управление из окон консоли и каждого консольного приложения (например, Cmd.exe, окна командной строки), осуществлявшего обмен данными с Csrss. Теперь Windows использует для каждого окна консоли, имеющего в системе отдельный процесс — хост окна консоли, — console window host (Conhost.exe). Одно и то же окно консоли может совместно использоваться несколькими консольными приложениями, как при запуске одной командной строки из другой командной строки. По умолчанию вторая командная строка совместно использует окно консоли первой командной строки.

Всякий раз, когда консольное приложение регистрируется с помощью экземпляра подсистемы Csrss, запущенной в текущем сеансе, Csrss создает новый экземпляр Conhost, используя вместо системного маркера Csrss маркер доступа клиентского процесса. Затем подсистема отображает на хосты окна консоли используемый ею общий раздел памяти, чтобы позволить всем хостам Conhost совместно использовать часть их памяти с Csrss

для эффективной обработки буфера (их потоки уже не существуют внутри Csrss), и создает поименованный порт асинхронного вызова локальной процедуры (Asynchronous Local Procedure Call, ALPC) в каталоге объектов управления \RPC (см. главу 3.) Имя порту дается в формате *console-PID-lpc-handle*, где PID является идентификатором Conhost-процесса. Затем происходит регистрация этого PID со структурой процесса ядра, связанной с пользовательским приложением, которое затем может запрашивать эту информацию для открытия только что созданного ALPC-порта. Этот процесс также задает соответствие объекта общего раздела памяти между приложением командной строки и его Conhost, чтобы они могли обмениваться данными. И наконец, в каталоге BaseNamedObjects сеанса 0 создается событие ожидания (которое называется ConsoleEvent-PID), чтобы приложение командной строки и Conhost смогли уведомить друг друга о новых данных буфера. На следующем рисунке показывается процесс Conhost с дескрипторами, открытыми для его ALPC-порта и события.



Поскольку Conhost работает с полномочиями пользователя (под которыми также подразумевается уровень прав пользователя), так же как в процессе, связанном с самим приложением консоли, процессы консоли защищаются механизмом безопасности, который называется «изоляцией привилегий пользовательского интерфейса» (User Interface Privilege Isolation). (UIPI-механизм описан в главе 6 «Безопасность».) Кроме этого консольные приложения, ограниченные возможностями центрального процессора, могут быть идентифицированы тем хост-процессом консоли, который их поддерживает (и который пользователь при необходимости может завершить). Поскольку теперь Conhost-процессы запускаются за пределами специального анклава подсистемы Csrss, возникает побочный эффект, благодаря которому консольные приложения (чьими окнами фактически владеет Conhost) могут быть полностью настроены на тему, загружать DLL-библиотеки сторонних производителей и запускаться с полноценными оконными возможностями.

Подсистема для приложений на Unix-основе

Подсистема для приложений на основе Unix — Subsystem for UNIX-based Applications (SUA) — позволяет компилировать и запускать пользовательские

приложения на UNIX-основе на компьютере под управлением серверной операционной системы Windows или клиентской операционной системы Windows версий Enterprise или Ultimate. SUA предоставляет около 2000 UNIX-функций и 300 UNIX-подобных средств и утилит. (Дополнительную информацию, касающуюся SUA, можно найти на сайте http://technet.microsoft.com/en-us/library/cc771470.aspx.) Дополнительная информация о том, как Windows обрабатывает запущенные SUA-приложения, дана в главе 5, в разделе «Порядок работы функции CreateProcess».

ИСХОДНАЯ ПОДСИСТЕМА POSIX

POSIX — широко известный акроним, означающий «a portable operating system interface based on UNIX» (интерфейс переносимой операционной системы на основе UNIX), ссылающийся на коллекцию международных стандартов для интерфейсов операционных систем UNIX-стиля. Стандарты POSIX подстегивают поставщиков, реализующих интерфейсы UNIX-стиля, сделать их совместимыми, чтобы программисты могли легко перемещать свои приложения из одной системы в другую.

Изначально в Windows был реализован только один из многих POSIX-стандартов, POSIX.1, официально известный как стандарт ISO/IEC 9945-1:1990 или IEEE POSIX standard 1003.1-1990. Этот стандарт был включен главным образом для того, чтобы соответствовать набору требований о государственных закупках США во второй половине 1980-х годов. От POSIX.1 требовалось соответствовать федеральному стандарту обработки информации — Federal Information Processing Standard (FIPS) 151-2, разработанному национальным институтом стандартов и технологии — National Institute of Standards and Technology. Windows NT 3.5, 3.51 и 4 были официально протестированы и сертифицированы как соответствующие стандарту FIPS 151-2.

Поскольку соответствие POSIX.1 было для Windows обязательной задачей, операционная система была разработана с гарантированным присутствием требуемой базовой системной поддержки, позволяющей реализовать подсистему POSIX.1 (это касается функции `fork`, реализованной в исполняющей системе Windows, и поддержки жестких ссылок на файлы в файловой системе Windows).

Ntdll.dll

Ntdll.dll является специальной библиотекой системной поддержки, предназначенной, главным образом, для использования DLL-библиотек подсистем. В ней содержатся функции двух типов:

- функции-заглушки, обеспечивающие переходы от диспетчера системных служб к системным службам исполняющей системы Windows;
- вспомогательные внутренние функции, используемые подсистемами, DLL-библиотеками подсистем и другими исходными образами.

Первая группа функций предоставляет интерфейс к службам исполняющей системы Windows, которые могут быть вызваны из пользовательского режима. К этой группе относятся более чем 400 функций, среди которых `NtCreateFile`, `NtSetEvent` и т. д. Как уже отмечалось, основная часть возможностей, присущих

данным функциям, доступна через Windows API. Но некоторые возможности недоступны и предназначены для использования только внутри операционной системы.

Для каждой из этих функций в Ntdll содержится точка входа с именем, совпадающим с именем функции. Код внутри функции содержит зависящую от конкретной архитектуры инструкцию, осуществляющую переход в режим ядра для вызова диспетчера системных служб (более подробно этот вопрос рассмотрен в главе 3), который после проверки ряда параметров вызывает настоящую системную службу режима ядра, реальный код которой содержится в файле Ntoskrnl.exe.

Ntdll также содержит множество вспомогательных функций, таких как загрузчики образов (префикс `Ldr`), диспетчер динамической области памяти и функции обмена данными между процессами подсистемы Windows (префикс `Csr`). Ntdll включает также общие подпрограммы библиотеки времени выполнения (префикс `Rtl`), поддержку отладки в пользовательском режиме (префикс `DbgUi`) и отслеживания событий для Windows (Event Tracing for Windows) (префикс `Etw`), а также диспетчер асинхронных вызовов процедур и исключений пользовательского режима (APC). (APC-вызовы и исключения рассматриваются в главе 3.) И наконец, в Ntdll находится небольшой поднабор подпрограмм времени выполнения языка C (CRT), ограниченный подпрограммами, являющимися частью строковых и стандартных библиотек (в качестве примера можно привести подпрограммы `memcpy`, `strcpy`, `itoa` и т. д.).

Исполняющая система

Исполняющая система Windows находится на верхнем уровне файла Ntoskrnl.exe. (Ядро составляет его нижний уровень.) Она включает функции следующих типов:

- Функции, экспортируемые и вызываемые из пользовательского режима. Эти функции называются системными службами и экспортируются посредством Ntdll. Большинство служб доступно через Windows API или через API-интерфейсы других подсистем среды окружения. Но часть служб не доступна ни через какие документированные функции подсистем. (В качестве примера можно привести ALPC и различные функции запросов, например `NtQueryInformationProcess`, специализированные функции, такие как `NtCreatePagingFile` и т. д.).
- Функции драйверов устройств, вызываемые с помощью функции `DeviceIoControl`, которая предоставляет общий интерфейс из пользовательского режима к режиму ядра для вызова тех функций в драйверах устройств, которые не связаны с чтением или записью.
- Функции, которые могут быть вызваны только из режима ядра, экспортируемые из WDK и документированные в этом инструментальном наборе.
- Функции, экспортируемые и вызываемые из режима ядра, но недокументированные в WDK (например, функции, вызываемые загрузочным видеодрайвером, чьи имена начинаются с префикса `Inbv`).
- Функции, определенные в качестве глобальных символов, но при этом не подлежащие экспорту. К их числу относятся внутренние вспомогательные функции, вызываемые внутри Ntoskrnl, например такие функции, чьи имена начинаются

с префикса `Iop` (внутренние функции поддержки диспетчера ввода-вывода) или с префикса `Mi` (внутренние функции поддержки диспетчера памяти).

- Функции, являющиеся внутренними по отношению к модулю, но не определенные в качестве глобальных символов.

Исполняющая система содержит следующие основные компоненты, каждый из которых подробно рассмотрен в следующих главах данной книги:

- *Диспетчер конфигурации* (см. главу 4), который отвечает за реализацию и управление системным реестром.
- *Диспетчер процессов* (см. главу 5) создает процессы и потоки и завершает их работу. Исходная поддержка процессов и потоков реализована в ядре Windows; исполняющая система добавляет к этим низкоуровневым объектам дополнительную семантику и функции.
- *Монитор безопасности* (security reference monitor, SRM) (см. главу 6) обеспечивает соблюдение политики безопасности на локальном компьютере. Он охраняет ресурсы операционной системы, выполняя защиту и проверку объектов времени выполнения.
- *Диспетчер ввода-вывода* реализует аппаратно-независимый ввод-вывод и отвечает за направление на соответствующие драйверы устройств для дальнейшей обработки.
- *Диспетчер устройств plug and play* (PnP) определяет, какие драйверы требуются для поддержки конкретного устройства, и загружает эти драйверы. В процессе переписи устройств для каждого из них извлекаются требования к аппаратным ресурсам. На основе требований к ресурсам каждого устройства диспетчер PnP назначает соответствующие аппаратные ресурсы, такие как порты ввода-вывода, линии запроса на прерывание (IRQ), DMA-каналы и адреса памяти. Он также отвечает за отправку соответствующих уведомлений о событиях при изменениях в устройствах (добавлении или удалении устройства) в системе.
- *Диспетчер электропитания* согласовывает события электропитания и генерирует уведомления ввода-вывода, касающиеся управления электропитания, посылаемые драйверам устройств. При простое системы диспетчер электропитания может быть настроен на снижение расхода электроэнергии путем перевода центрального процессора в спящий режим. Изменения в энергопотреблении отдельными устройствами управляются драйверами устройств, но согласовываются диспетчером электропитания.
- *Подпрограммы инструментария управления Windows для модели драйверов — Windows Driver Model Windows Management Instrumentation routines* (см. главу 4) позволяют драйверам устройств публиковать информацию о производительности и конфигурации, а также получать команды от WMI-службы пользовательского режима. Потребители WMI-информации могут быть на локальной машине или на удаленной, имеющей сетевой доступ
- *Диспетчер кэша* повышает производительность файлового ввода-вывода, размещая данные, полученные с диска, к которым недавно было обращение, в основной памяти для ускорения доступа к этим данным (и задерживая записи на диск путем кратковременного хранения обновлений в памяти перед их отправкой на диск). Вы увидите, что это делается путем поддержки диспетчером памяти отображаемых файлов.

- *Диспетчер памяти* реализует виртуальную память, схему управления памятью, предоставляющую каждому процессу большое, закрытое адресное пространство, которое может превышать по объему доступную физическую память. Диспетчер памяти также предоставляет исходную поддержку диспетчеру кэша.
- *Логическая предвыборка* и *Superfetch* ускоряют работу системы и запуск процесса путем оптимизации загрузки данных, на которые есть ссылка во время запуска системы или процесса.

Помимо этого исполняющая система содержит четыре основные группы вспомогательных функций, используемых только что перечисленными исполняющими компонентами. Около трети этих вспомогательных функций документированы в WDK, поскольку они также используются драйверами устройств. Есть четыре категории вспомогательных функций:

- *Диспетчер объектов*, который работает с исполняющими объектами Windows и абстрактными типами данных, используемыми для представления таких ресурсов операционной системы, как процессы, потоки и различные объекты синхронизации, создает эти объекты, управляет ими и удаляет их. Диспетчер объектов рассматривается в главе 3.
- *Усовершенствованная система вызова локальных процедур* (ALPC, рассматриваемая в главе 3) передает сообщения между клиентским и серверным процессами на одном и том же компьютере. Кроме всего прочего, ALPC используется в качестве локального средства сообщения для вызова удаленных процедур (remote procedure call, RPC), стандартного средства связи для клиентских и серверных процессов по сети.
- Широкий набор *библиотечных функций времени выполнения* общего назначения, среди которых функции обработки строк, арифметических операций, преобразований типов данных и работы со структурой безопасности.
- Вспомогательные подпрограммы исполняющей системы, к которым относятся подпрограммы распределения системной памяти (выгружаемого и невыгружаемого пула), доступа к памяти с взаимной блокировкой, а также три специальных типа объектов синхронизации: ресурсы (resources), быстрые мьютексы (fast mutexes) и пуш-блокировки (pushlocks).

Исполняющая система также содержит различные инфраструктурные подпрограммы (некоторые из них будут кратко упомянуты в данной книге):

- Библиотеку *отладчика ядра*, позволяющую вести отладку ядра из поддерживающего отладку KD, переносимого протокола, поддерживаемого через различные средства транспортировки данных (такие как USB и IEEE 1394) и реализованного в утилитах WinDbg и Kd.exe.
- *Среду отладки в режиме пользователя*, которая отвечает за отправку событий API-интерфейсу отладки в режиме пользователя и позволяющую расставлять контрольные точки и проводить пошаговое выполнение кода для его отработки, а также для изменения контекста запущенных потоков.
- *Диспетчер транзакций ядра*, предоставляющий диспетчерам ресурсов простой, двухфазный механизм выделения ресурсов, такой как транзакционный реестр —(TxR) и транзакционная NTFS (TxF).

- *Библиотеку гипервизора*, часть стека Hyper-V в Windows Server 2008, предоставляющая на уровне ядра поддержку среды виртуальных машин и оптимизирующая определенные части кода, когда системе известно, что она работает в клиентском разделе (виртуальной среде).
- *Диспетчер исправлений*, предоставляющий пути обхода для нестандартных или несовместимых аппаратных устройств.
- *Средство проверки драйверов* — Driver Verifier, реализующее дополнительные проверки целостности кода и драйверов режима ядра.
- *Средство трассировки событий* — Event Tracing for Windows, предоставляющее вспомогательные подпрограммы для трассировки общесистемных событий для компонентов режима ядра и пользовательского режима.
- *Инфраструктуру диагностики Windows*, допускающую интеллектуальное отслеживание деятельности системы на основе сценариев диагностики.
- *Подпрограммы поддержки архитектуры аппаратных ошибок Windows*, предоставляющие общую среду для отчетов об ошибках оборудования.
- *Библиотеку времени выполнения файловой системы*, предоставляющую подпрограммы общей поддержки для драйверов файловой системы.

Ядро

Ядро состоит из набора функций, находящихся в файле Ntoskrnl.exe. Этот набор предоставляет основные механизмы (службы диспетчеризации потоков и синхронизации), используемые компонентами исполняющей системы, а также поддержкой архитектурно-зависимого оборудования низкого уровня (например, диспетчеризацией прерываний и исключений), которое имеет отличия в архитектуре каждого процессора. Код ядра написан главным образом на C, с ассемблерным кодом, предназначенным для задач, требующих доступа к специальным инструкциям и регистрам процессора, доступ к которым из кода на языке C затруднен.

Подобно различным вспомогательным функциям, упомянутым в предыдущем разделе, ряд функций ядра документированы в WDK (и их описание может быть найдено при поиске функций, начинающихся с префикса `Ke`), поскольку они нужны для реализации драйверов устройств.

Объекты ядра

Ядро предоставляет низкоуровневую базу из четко определенных, предсказуемых примитивов и механизмов операционной системы, позволяющую высокоуровневым компонентам исполняющей системы выполнять свои функции. Само ядро отделено от остальной исполняющей системы путем реализации механизмов операционной системы и уклонения от выработки политики. Оно оставляет почти все политические решения, за исключением планирования и диспетчеризации потоков, реализуемых ядром, за исполняющей системой.

За пределами ядра исполняющая система представляет потоки и другие ресурсы совместного использования в виде объектов. Эти объекты требуют некоторых издержек, например на дескрипторы для управления ими, на проверки безопасности для их защиты и на ресурсные квоты, выделяемые при их создании.

В ядре, реализующем набор менее сложных объектов, называемых «объектами ядра», подобные издержки исключены, что помогает ядру управлять основной обработкой и поддерживать создание исполняющих объектов. Большинство объектов уровня исполнения инкапсулируют один или несколько объектов ядра, принимая их, определенные в ядре свойства.

Один набор объектов ядра, называемых «управляющими объектами», определяет семантику управления различными функциями операционной системы. В этот набор включены объекты асинхронного вызова процедур — APC, отложенного вызова процедур — deferred procedure call (DPC), и несколько объектов, используемых диспетчером ввода-вывода, например объект прерывания.

Еще один набор объектов ядра, известных как «объекты-диспетчеры», включает возможности синхронизации, изменяющие или влияющие на планирование потоков. Объекты-диспетчеры включают поток ядра, мьютекс (называемый среди специалистов «мутантом»), событие, пару событий ядра, семафор, таймер и таймер ожидания. Исполняющая система использует функции ядра для создания экземпляров объектов ядра, работы с ними и создания более сложных объектов, предоставляемых в пользовательском режиме. Более подробно объекты рассматриваются в главе 3, а процессы и потоки рассматриваются в главе 5.

Область ядра, относящаяся к управлению процессором, и блок управления (KPCR и KPRCB)

Для хранения специфических для процессора данных ядром используется структура данных, называемая областью, относящейся к управлению процессором, или KPCR (Kernel Processor Control Region). KPCR содержит основную информацию, такую как процессорная таблица диспетчеризации прерываний (interrupt dispatch table, IDT), сегмент состояния задачи (task-state segment, TSS) и таблица глобальных дескрипторов (global descriptor table, GDT). Она также включает состояние контроллера прерываний, которое используется вместе с другими модулями, такими как ACPI-драйвер и HAL. Для обеспечения простого доступа к KPCR ядро хранит указатель на эту область в регистре `fs` на 32-разрядной системе Windows и в регистре `gs` на Windows-системе x64. На системах IA64 KPCR всегда находится по адресу `0xe0000000ffff0000`.

KPCR также содержит вложенную структуру данных, которая называется блоком управления процессором (kernel processor control block, KPRCB). В отличие от области KPCR, которая документирована для драйверов сторонних производителей и для других внутренних компонентов ядра Windows, KPRCB является закрытой структурой, используемой только кодом ядра, который находится в файле Ntoskrnl.exe. В этом блоке содержится:

- информация о планировании (такая как текущий, следующий и приостановленный потоки, предназначенные для выполнения на процессоре);
- предназначенная для процессора база данных диспетчера (включающая готовые очереди для каждого приоритетного уровня);
- DPC-очередь;
- информация о производителе центрального процессора и идентификационная информация (модель, степпинг, скорость, биты особенностей);

- информация о топологии центрального процессора и о технологии доступа к неоднородной памяти — NUMA (информация об узле, о количестве логических процессоров в каждом ядре и т. д.);
- информация о размерах кэш-памяти;
- информация об учете времени (такая как время DPC и обработки прерывания) и многое другое.

В KPRCB также содержится вся статистика процессора, такая как статистика ввода-вывода, статистика диспетчера кэша, статистика DPC и статистика диспетчера памяти. И наконец, KPRCB иногда используется для хранения структур выравнивания границ кэша для каждого процессора, необходимых для оптимизации доступа к памяти, особенно на NUMA-системах. Например, система невыгружаемого и выгружаемого пула со стороны выглядит как списки, хранящиеся в KPRCB.

ЭКСПЕРИМЕНТ: ПРОСМОТР KPCR И KPRCB

Содержимое KPCR и KPRCB можно просмотреть, используя команды отладки ядра `!pcr` и `!prcb`. Без пометок отладчик по умолчанию покажет информацию для центрального процессора 0; но вы можете указать центральный процессор, добавив после команды его номер (например, `!pcr 2`). В следующем примере показано, как выглядит вывод команд `!pcr` и `!prcb`. Если в системе имелись задержанные DPC-вызовы, эта информация также будет отображена на экране.

```
lkd> !pcr
KPCR for Processor 0 at 81d09800:
    Major 1 Minor 1
    NtTib.ExceptionList: 9b31ca3c
        NtTib.StackBase: 00000000
       NtTib.StackLimit: 00000000
     NtTib.SubSystemTib: 80150000
          NtTib.Version: 1c47209e
      NtTib.UserPointer: 00000001
          NtTib.SelfTib: 7ffde000
                SelfPcr: 81d09800
                   Prcb: 81d09920
                   Irql: 00000002
                    IRR: 00000000
                    IDR: ffffffff
          InterruptMode: 00000000
                    IDT: 82fb8400
                    GDT: 82fb8000
                    TSS: 80150000
          CurrentThread: 86d317e8
             NextThread: 00000000
             IdleThread: 81d0d640
              DpcQueue:
lkd> !prcb
PRCB for Processor 0 at 81d09920:
Current IRQL -- 0
```

```
Threads--  Current 86d317e8 Next 00000000 Idle 81d0d640
Number 0 SetMember 1
Interrupt Count -- 294ccce0
Times -- Dpc    0002a87f Interrupt 00010b87
         Kernel 026270a1 User      00140e5e
```

Для непосредственного вывода дампа структур данных _KPCR и _KPRCB можно воспользоваться командой dt, поскольку обе команды отладки дают вам адреса структур (которые для наглядности выделены в предыдущем выводе жирным шрифтом). Например, если нужно определить скорость процессора, можно с помощью следующей команды посмотреть на поле MHz:

```
lkd> dt nt!_KPRCB 81d09920 MHz
   +0x3c4 MHz : 0xbb4
lkd> ? bb4
Evaluate expression: 2996 = 00000bb4
```

На данной машине процессор был запущен на частоте около 3 ГГц.

Поддержка оборудования

Другой основной задачей ядра является абстрагирование или изоляция исполняющей системы и драйверов устройств от различий аппаратных архитектур, поддерживаемых Windows. Эта задача включает обработку вариантов в таких функциях, как обработка прерываний, диспетчеризация исключений и мультипроцессорная синхронизация.

Даже для этих, зависящих от аппаратных особенностей, функций конструкция ядра пытается максимизировать объем общего кода. Ядро поддерживает набор переносимых и семантически идентичных для разных архитектур интерфейсов. Основная часть кода, реализующего эти переносимые интерфейсы, также идентична для разных архитектур.

Часть этих интерфейсов для разных архитектур реализована по-разному или частично реализована с использованием кода, специфичного для конкретной архитектуры. Такие, независимые от архитектуры, интерфейсы могут быть вызваны на любой машине, и семантические характеристики интерфейса будут одинаковыми, независимо от различий в коде, связанных с архитектурой. Некоторые интерфейсы ядра (см. подпрограммы спин-блокировки в главе 3) в действительности реализованы на уровне аппаратных абстракций — HAL, поскольку их реализация может варьироваться для систем внутри одного и того же архитектурного семейства.

Ядро также содержит небольшой объем кода с интерфейсами для x86-систем, необходимыми для поддержки старых программ MS-DOS. Эти x86-интерфейсы не обладают переносимостью, в том смысле, что они не могут вызываться на машинах, основанных на каких-нибудь других архитектурах; их там просто не будет. Этот код, предназначенный для x86-систем, к примеру, поддерживает вызовы для обращения с таблицами глобальных дескрипторов (GDT) и таблицами локальных дескрипторов (LDT), являющихся аппаратными особенностями x86.

Другие примеры кода ядра, зависящего от конкретной архитектуры, включают интерфейсы для обеспечения поддержки буфера трансляции и кэш-памяти

центрального процессора. Из-за способов реализации кэш-памяти эта поддержка требует применения разного кода для разных архитектур.

Другим примером может послужить переключение контекста. Хотя на высоком уровне для выбора потока и переключения контекста используется один и тот же алгоритм (контекст предыдущего потока сохраняется, загружается контекст нового потока и запускается новый поток), в реализациях на разных процессорах существуют архитектурные различия. Поскольку контекст описывается состоянием процессора (его регистров и т. д.), сохраняемый и загружаемый контекст варьируется в зависимости от архитектуры.

Уровень аппаратных абстракций

Как уже упоминалось в начале главы, одним из наиболее важных элементов конструкции Windows является ее переносимость между разнообразными аппаратными платформами. Уровень аппаратных абстракций — hardware abstraction layer (HAL) является ключевой частью, обеспечивающей возможность такой переносимости. HAL является загружаемым модулем режима ядра (Hal.dll), обеспечивающим низкоуровневый интерфейс с аппаратной платформой, на которой запущена Windows. Он скрывает подробности, зависящие от аппаратуры, такие как интерфейсы ввода-вывода, контроллеры прерываний и механизмы взаимодействия процессоров, — любые функции, имеющие как архитектурные, так и машинные зависимости.

Поэтому вместо непосредственного доступа к оборудованию, внутренние компоненты Windows, а также написанные пользователями драйверы устройств, при необходимости получения информации, зависящей от платформы, поддерживают переносимость путем вызова HAL-подпрограмм. По этой причине HAL-подпрограммы документированы в WDK. Для получения дополнительной информации о HAL и его использовании драйверами устройств нужно обратиться к WDK.

Хотя в операционную систему включено несколько HAL-модулей (см. табл. 2.4), у Windows есть возможность определить во время загрузки, какой HAL-модуль должен использоваться, исключая проблемы, существовавшие в ранее выпущенных версиях Windows при попытке загрузки установки Windows на разных типах систем.

Таблица 2.4. Перечень HAL-модулей для x86

Имя HAL-файла	Поддерживаемые системы
Halacpi.dll	Персональные компьютеры с усовершенствованным интерфейсом управления конфигурированием и энергопотреблением — Advanced Configuration and Power Interface (ACPI). Предназначается только для однопроцессорной машины без поддержки усовершенствованного программируемого контроллера прерываний — APIC (наличие любого из таких контроллеров заставит систему использовать вместо этого HAL-модуль, показанный ниже)
Halmacpi.dll	Персональные компьютеры с усовершенствованным программируемым контроллером прерываний — Advanced Programmable Interrupt Controller (APIC), имеющие ACPI. Наличие APIC подразумевает поддержку симметричной мультипроцессорной обработки — SMP

ПРИМЕЧАНИЕ

На x64-машинах имеется только один HAL-образ по имени Hal.dll. Это обусловлено наличием у всех x64-машин материнских плат одинаковой конфигурации, поскольку процессы требуют поддержки ACPI и APIC. Следовательно, поддержка машин без ACPI или со стандартного программируемого контроллера прерываний — PIC, не требуется.

ЭКСПЕРИМЕНТ: ОПРЕДЕЛЕНИЕ ЗАПУЩЕННОГО HAL-МОДУЛЯ

Определить, какая версия HAL-модуля запущена, можно с помощью WinDbg и открытия сеанса локальной отладки ядра. Обеспечьте путем ввода команды `.reload` загрузку символов, а затем наберите команду `lm vm hal`. Например, следующий вывод получен на системе, запустившей ACPI HAL:

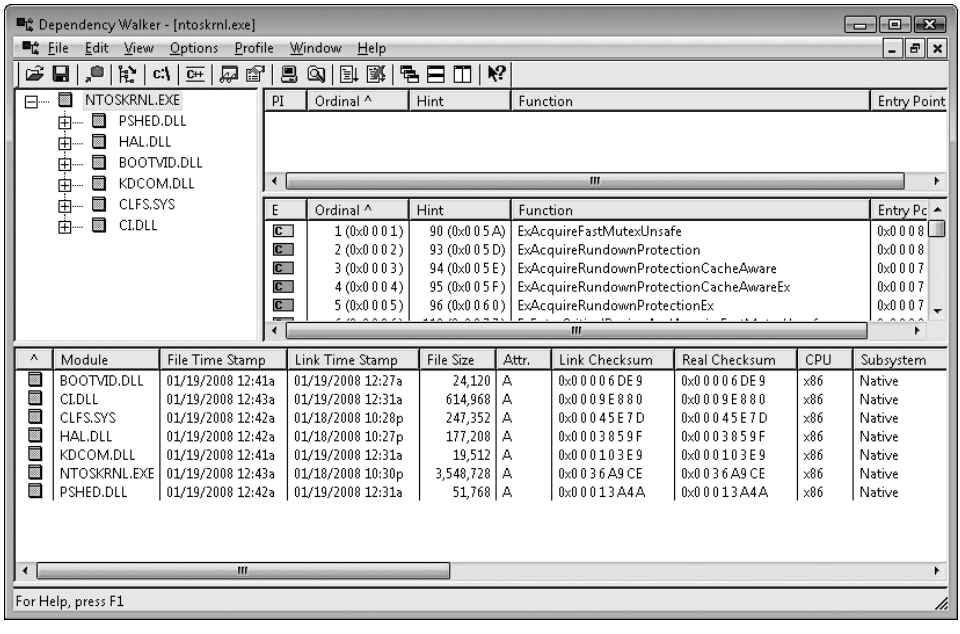
```
lkd> lm vm hal
start    end        module name
fffff800'0181b000 fffff800'01864000   hal        (deferred)
    Loaded symbol image file: halmacpi.dll
    Image path: halmacpi.dll
    Image name: halmacpi.dll
    Timestamp:        Mon Jul 13 21:27:36 2009 (4A5BDF08)
    CheckSum:         0004BD36
    ImageSize:        00049000
    File version:     6.1.7600.16385
    Product version:  6.1.7600.16385
    File flags:       0 (Mask 3F)
    File OS:          40004 NT Win32
    File type:        2.0 Dll
    File date:        00000000.00000000
    Translations:     0409.04b0
    CompanyName:      Microsoft Corporation
    ProductName:      Microsoft® Windows® Operating System
    InternalName:     halmacpi.dll
    OriginalFilename: halmacpi.dll
    ProductVersion:   6.1.7600.16385
    FileVersion:      6.1.7600.16385 (win7_rtm.090713-1255)
    FileDescription:  Hardware Abstraction Layer DLL
    LegalCopyright:   © Microsoft Corporation. All rights reserved.
```

■

ЭКСПЕРИМЕНТ: ПРОСМОТР ЗАВИСИМОСТЕЙ NTOSKRNL И HAL

Взаимоотношения между ядром и HAL-образами можно просмотреть путем изучения их таблиц экспорта и импорта с помощью средства Dependency Walker (Depends.exe). Для изучения образа в Dependency Walker выберите пункт Open (Открыть) в меню File (Файл), чтобы открыть требуемый файл образа.

Пример вывода, который можно увидеть путем просмотра зависимостей Ntoskrnl с использованием этого средства может иметь следующий вид.



Обратите внимание на то, что Ntoskrnl связан с HAL, который, в свою очередь, связан с Ntoskrnl. (Они оба используют функции друг друга.) Ntoskrnl также связан со следующими исполняемыми файлами:

- Pshed.dll, драйвер ошибок оборудования — Platform-Specific Hardware Error Driver (PSHED), зависящий от используемой платформы. Этот драйвер предоставляет абстракцию средства выдачи отчетов об ошибках оборудования исходной платформы, скрывая подробности присущих платформе механизмов обработки ошибок от операционной системы и открывая операционной системе Windows однообразный интерфейс.
- Bootvid.dll (только на 32-разрядных операционных системах), видеодрайвер загрузки. Bootvid предоставляет поддержку команд VGA, требуемых для отображения при запуске загрузочного текста и загрузочного логотипа. На системах x64 эта библиотека во избежание конфликтов со средством защиты ядра от исправлений — Kernel Patch Protection (KPP) встроена в ядро. (Более подробно информация о KPP и PatchGuard дана в главе 3.)
- Kdcom.dll, библиотека протокола коммуникаций отладчика ядра — Kernel Debugger Protocol (KD) Communications Library.
- Ci.dll, библиотека целостности кода. (Дополнительная информация о целостности кода дана в главе 3.)
- Clfs.sys, драйвер общей системы файлов журнала, используемых, кроме всего прочего, диспетчером транзакций ядра — Transaction Manager (KTM). (Дополнительная информация о KTM дана в главе 3.)

Подробное описание информации, показываемой этим средством, дана в справочном файле по Dependency Walker (Depends.hlp). ■

Драйверы устройств

В этом разделе предоставляется краткий обзор типов драйверов и объясняется, как получить список драйверов, установленных и загруженных на вашей системе.

Драйверы устройств являются загружаемыми модулями режима ядра (имена файлов которых обычно имеют расширение .sys), обеспечивающими интерфейс между диспетчером ввода-вывода и соответствующим оборудованием. Они запускаются в режиме ядра в одном их трех контекстов:

- в контексте пользовательского потока, инициировавшего функцию ввода-вывода;
- в контексте системного потока режима ядра;
- в результате прерывания (и поэтому вне контекста любого конкретного процесса или потока — какой бы процесс или поток ни был текущим при возникновении прерывания).

Как утверждалось в предыдущем разделе, драйверы устройств в Windows не работают с оборудованием напрямую, а вызывают для получения интерфейса с оборудованием функции в HAL. Драйверы обычно пишутся на C (иногда на C++) и поэтому, при правильном использовании подпрограмм HAL могут переноситься в виде исходного кода между архитектурами центральных процессоров, поддерживаемых Windows, а в виде исполняемого файла могут переноситься внутри архитектурного семейства.

Существует несколько типов драйверов устройств:

- *Драйверы аппаратных устройств*, управляющие (с помощью HAL) записью на физическое устройство или в сеть или чтением оттуда. Существует множество типов драйверов аппаратных устройств — драйверы шин, драйверы интерфейса с пользователем, драйверы запоминающих устройств большой емкости и т. д.
- *Драйверы файловой системы*, являющиеся драйверами Windows, которые принимают запросы на файловый ввод-вывод и транслируют их в запросы ввода-вывода, направляемые конкретному устройству.
- *Драйверы фильтра файловой системы*, обеспечивающие зеркалирование и шифрование дисков, перехваты ввода-вывода и некоторую дополнительную обработку перед передачей ввода-вывода на следующий уровень.
- *Сетевые редиректоры и серверы*, являющиеся драйверами файловой системы, передающими запросы ввода-вывода файловой системы машине, находящейся в сети, и получающими от нее аналогичные запросы.
- *Драйверы протоколов*, реализующие такие сетевые протоколы, как TCP/IP, NetBEUI и IPX/SPX.
- *Драйверы потоковых фильтров ядра*, собранные в цепочку для обработки сигналов потока данных, например, при записи или воспроизведении аудио- и видеопотоков.

Поскольку единственным способом добавления к системе написанного пользователям кода режима ядра является установка драйвера устройства, некоторые программисты пишут драйверы устройств просто для получения способа

доступа к внутренним функциям или структурам данных операционной системы, недоступным из пользовательского режима (но документированным и поддерживаемым в WDK). Например, многие утилиты из Sysinternals сочетают в себе приложение Windows GUI и драйвер устройства, используемый для сбора внутреннего состояния системы и вызова функций, доступных только в режиме ядра и недоступных из Windows API в режиме пользователя.

Модель драйверов Windows (WDM)

В Windows 2000 была добавлена поддержка технологии Plug and Play, настроек электропитания и расширение модели драйверов Windows NT, названной моделью драйверов Windows (WDM). Windows 2000 и более поздние версии могут запускать драйверы, унаследованные у Windows NT 4, но, поскольку они не поддерживают технологию Plug and Play настройки электропитания, системы, запускающие эти драйверы, будут вынуждены ограничивать возможности в этих двух областях.

С точки зрения WDM, существуют драйверы трех типов:

- *Драйвер шины*, обслуживающий контроллер шины, адаптер, мост или любое устройство, имеющее дочерние устройства. Драйверы шины нуждаются в драйверах, и Microsoft, как правило, их предоставляет; каждый тип шины (такой как PCI, PCMCIA и USB), имеющийся в системе, имеет один драйвер шины. Сторонние производители могут создавать драйверы шины для предоставления поддержки новых шин, таких как VMEbus, Multibus и Futurebus.
- *Функциональный драйвер*, являющийся основным драйвером устройства и предоставляющий для него управляющий интерфейс. Драйвер нужен в том случае, если устройство не используется напрямую (в варианте реализации, при которой ввод-вывод осуществляется драйвером шины и любыми драйверами фильтра шины, в качестве примера можно привести SCSI PassThru). Функциональный драйвер по определению является драйвером, который знает о конкретном устройстве практически все, и обычно он является единственным драйвером, обращающимся к специфическим регистрам устройства.
- *Драйвер фильтра*, использующийся для добавления функциональности к устройству (или к существующему драйверу) или для изменения запросов ввода-вывода или ответов от других драйверов (для настройки оборудования, предоставляющего неверную информацию о требованиях к аппаратным ресурсам). Драйверы фильтра являются дополнительными и могут присутствовать в любом количестве, размещаясь выше или ниже функционального драйвера и выше драйвера шины. Обычно драйверы фильтра поставляются OEM-производителями или независимыми поставщиками оборудования (IHV).

В среде окружения WDM все аспекты устройства контролируются не одним драйвером: драйвер шины занимается отправкой диспетчеру PnP отчетов об устройствах, подключенных к его шине, а функциональный драйвер управляет самим устройством.

В большинстве случаев драйверы фильтра, находящиеся на нижнем уровне, изменяют поведение устройства. Например, если устройство сообщает своему драйверу шины, что ему нужно 4 порта ввода-вывода, в то время как ему факти-

чески нужно 16 портов ввода-вывода, функциональный драйвер фильтра для данного конкретного устройства может перехватить перечень аппаратных ресурсов, о котором драйвер шины сообщает диспетчеру PnP, и исправить количество портов ввода-вывода.

Драйверы фильтра, находящиеся на верхнем уровне, обычно предоставляют устройству какие-нибудь дополнительные свойства. Например, драйвер фильтра такого устройства, как клавиатура, находящийся на верхнем уровне, может навязывать дополнительные проверки безопасности.

Обработка прерывания рассматривается в главе 3.

Windows Driver Foundation

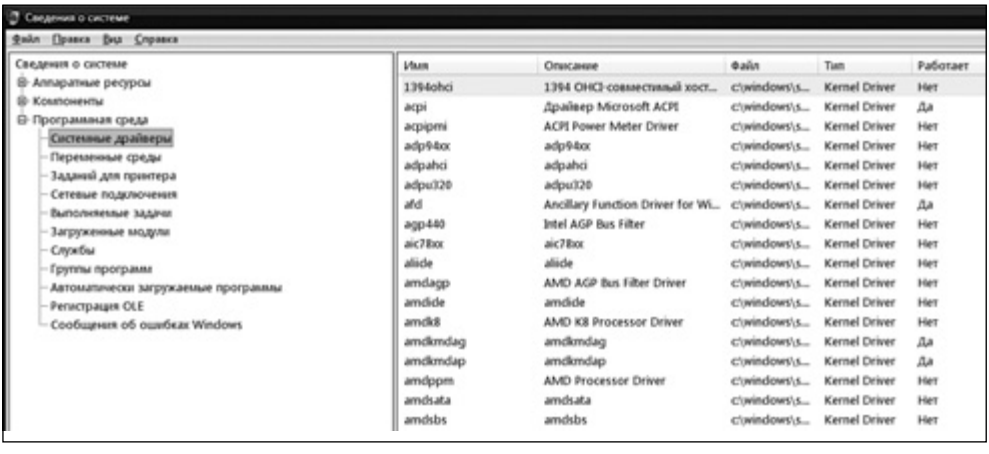
Набор инструментальных средств Windows Driver Foundation (WDF) упрощает разработку драйверов Windows, предоставляя для этого две среды: Kernel-Mode Driver Framework (KMDF) и User-Mode Driver Framework (UMDF). Разработчики могут использовать KMDF для написания драйверов для Windows 2000 SP4 и более поздних версий, в то время как UMDF поддерживает Windows XP и более поздние версии.

KMDF предоставляет простой интерфейс к WDM и скрывает все ее сложности от создателя драйвера, не изменяя исходной модели «шина-функция-фильтр». Драйверы KMDF отвечают за события, которые могут быть ими зарегистрированы, и осуществляют вызовы к библиотеке KMDF для выполнения работы, неспецифичной для управляемого ими оборудования, например для общего управления электропитанием или для синхронизации. (Ранее это должен был самостоятельно реализовывать каждый драйвер.) В некоторых случаях более чем 200 строк кода WDM можно заменить одним лишь вызовом функции KMDF.

UMDF позволяет создавать драйверы определенных классов (главным образом основанные на USB или других шинах, использующих протокол с большими задержками), например драйверы для видеокамер, MP3-плееров, мобильных телефонов, наладонников и принтеров, которые могут быть реализованы как драйверы пользовательского режима. По сути UMDF запускает каждый драйвер пользовательского режима в виде службы пользовательского режима и использует ALPC для связи с драйвером-упаковщиком режима ядра, предоставляющего фактический доступ к оборудованию. Если UMDF-драйвер попадает в аварийную ситуацию, процесс завершается и, как правило, перезапускается, поэтому система сохраняет стабильность — устройство просто становится недоступным, пока перезапускается служба, являющаяся хозяином драйвера. И наконец, UMDF-драйверы пишутся на C++ с использованием COM-подобных классов и семантики, тем самым еще больше понижая планку для программистов, пишущих драйверы устройств.

ЭКСПЕРИМЕНТ: ПРОСМОТР УСТАНОВЛЕННЫХ ДРАЙВЕРОВ УСТРОЙСТВ

Получить перечень установленных драйверов можно путем запуска Msinfo32. (Для запуска этого средства щелкните на кнопке Пуск (Start), наберите `Msinfo32` и нажмите клавишу Ввод.) В разделе Сведения о системе (System Summary) раскройте пункт Программная среда (Software Environment) и откройте окно Системные драйверы (System Drivers). Пример вывода перечня установленных драйверов имеет следующий вид.



В этом окне выведен перечень драйверов устройств, обнаруженных в реестре с указанием их типов и состояния — Running (работает) или Stopped (остановлен). И драйверы устройств, и процессы служб Windows определены в одном и том же месте: HKLM\SYSTEM\CurrentControlSet\Services. Но они различаются по типу кода, например тип 1 относится к драйверу устройства режима ядра. (Полный перечень информации, сохраненной в реестре для драйверов устройств, дан в табл. 4.7 в главе 4.)

Вместо этого перечень текущих загруженных драйверов устройств можно получить, выбрав System process в Process Explorer и открыв просмотр DLL. ■

ПРИСМАТРИВАЯСЬ К НЕДОКУМЕНТИРОВАННЫМ ИНТЕРФЕЙСАМ

Прояснить ситуацию можно путем изучения имен экспортированных или глобальных символов в ключевых системных образах (таких как Ntoskrnl.exe, Hal.dll или Ntdll.dll). Вы можете получить представление о тех вещах, которые может сделать Windows, а не только о том, что уже документировано и поддержано на данный момент. Разумеется, одно только знание имен этих функций еще не означает, что вы можете или должны их вызвать, — интерфейсы не документированы и могут подвергаться изменениям. Мы предлагаем вам только присмотреться к этим функциям для получения более полного представления о видах внутренних функций, выполняемых Windows, а не для обхода поддерживаемых интерфейсов.

Например, просмотр списка функций в Ntdll.dll даст вам возможность сравнить список всех системных служб, предоставляемых Windows DLL-библиотекам подсистем пользовательского режима, с подмножеством, показываемым каждой подсистемой. Хотя многие из этих функций отображаются на документированные и поддерживаемые Windows-функции, некоторые из них недоступны из Windows API. (См. статью «Inside the Native API» на сайте Sysinternals.)

С другой стороны, также интересно изучить то, что импортируют DLL-библиотеки подсистемы Windows (такие как Kernel32.dll или Advapi32.dll) и какие функции они вызывают в Ntdll.

Также интересно посмотреть на дамп файла Ntoskrnl.exe. Хотя многие экспортируемые подпрограммы, используемые драйверами устройств режима ядра, документированы в Windows Driver Kit, довольно многие из них остались недокументированными. Может также вызвать интерес просмотр таблицы импорта для Ntoskrnl и HAL; в этой таблице показан список функций в HAL, которые используются в Ntoskrnl, и наоборот.

В табл. 2.5 показан список большинства широко используемых префиксов имен функций компонентов исполняющей системы. Каждый из этих основных компонентов исполнительной системы также использует слегка измененные префиксы для обозначения внутренних функций — либо за первой буквой префикса следует буква *i* (означающая *internal*, внутренняя), либо за всем префиксом следует буква *p* (означающая *private*, закрытая). Например, *Ki* представляет внутренние функции ядра, а *Psp* ссылается на внутренние функции поддержки процесса.

Упростить расшифровку имен этих экспортируемых функций поможет понимание соглашения об именах системных подпрограмм Windows. Общий формат имеет следующий вид:

```
<Префикс><Операция><Объект>
```

В этом формате *Префикс* представляет внутренний компонент, экспортирующий подпрограмму, *Операция* сообщает о том, что будет сделано с объектом или ресурсом, а Объект идентифицирует то, над чем будет проводиться операция.

Например, `ExAllocatePoolWithTag` является вспомогательной процедурой исполняющей системы для выделения из выгружаемого или невыгружаемого пула. `KeInitializeThread` является процедурой, которая назначает и создает объект ядра «поток».

Таблица 2.5. Часто используемые префиксы

Префикс	Компонент
Alpc	Расширенное локальное межпроцессное взаимодействие
Cc	Общая кэш-память
Cm	Диспетчер конфигурации
Dbgk	Среда отладки для пользовательского режима
Em	Диспетчер исправлений
Etw	Отслеживание событий для Windows
Ex	Подпрограммы поддержки исполняющей системы
FsRtl	Библиотека времени выполнения драйвера файловой системы
Hvl	Библиотека гипервизора
Io	Диспетчер ввода-вывода
Kd	Отладчик ядра
Ke	Ядро
Lsa	Авторизация локальных пользователей
Mm	Диспетчер памяти

Таблица 2.5 *(продолжение)*

Префикс	Компонент
Nt	Системные службы NT (большинство из которых экспортируется в качестве Windows-функций)
Ob	Диспетчер объектов
Pf	Prefetcher
Po	Диспетчер электропитания
Pp	Диспетчер PnP
Ps	Поддержка процессов
Rtl	Библиотека времени выполнения
Se	Безопасность
Sm	Диспетчер запоминающего устройства
Tm	Диспетчер транзакций
Vf	Верификатор
Wdi	Инфраструктура диагностики Windows
Whea	Архитектура ошибок оборудования Windows
Wmi	Инструментарий управления Windows
Zw	Зеркальная точка входа для системных служб (имена которых начинаются с Nt), которая устанавливает предыдущий режим доступа к ядру, что исключает проверку параметров, поскольку системные службы Nt проверяют параметры только в том случае, если предыдущий режим доступа был пользовательским

Системные процессы

В каждой системе Windows появляются следующие системные процессы (Idle и System не являются полноценными процессами, поскольку в них не запускается какой-нибудь исполняемый код пользовательского режима):

- процесс Idle (содержащий по одному потоку на каждый центральный процессор для подсчета времени его простоя);
- процесс System (содержащий основную часть системных потоков режима ядра);
- диспетчер сеанса (Smss.exe);
- диспетчер локальных сеансов (Lsm.exe);
- подсистема Windows (Csrss.exe);
- инициализация сеанса 0 (Wininit.exe);
- процесс входа в систему (Winlogon.exe);
- диспетчер управления службами (Services.exe) и создаваемые им процессы дочерних служб (например, поддерживаемый системой универсальный сервис-хост процесс, Svchost.exe);
- сервер проверки подлинности локальной системы безопасности (Lsass.exe).

Чтобы понять взаимосвязь этих процессов, полезно будет просмотреть «дерево» процессов, то есть связь между родительскими и дочерними процессами. Эта связь поможет понять, откуда появляется тот или иной процесс. На рис. 2.5 показана копия экрана дерева процессов, просматриваемого после загрузочной трассировки, предпринятой средством Process Monitor. Использование средства Process Monitor позволяет видеть процессы, выход из которых на тот момент уже был осуществлен (помеченные блеклыми значками).

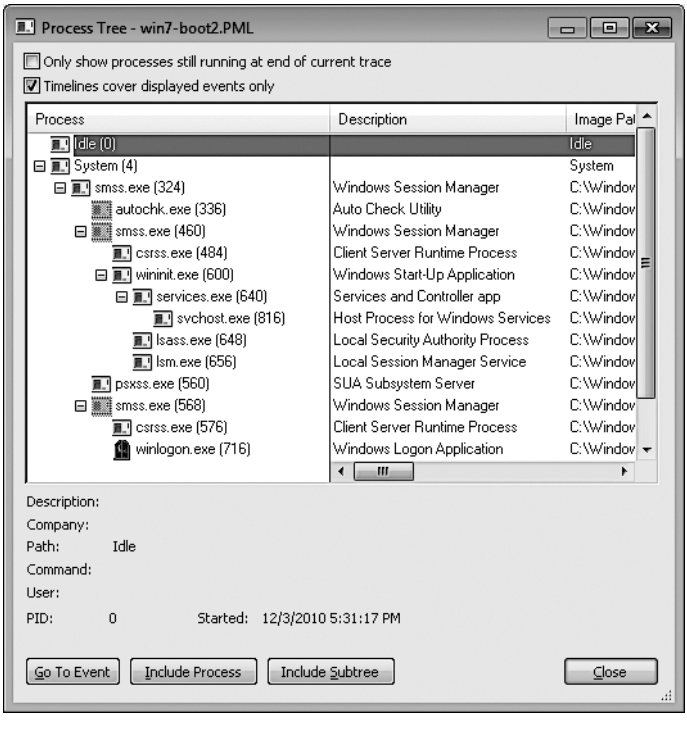


Рис. 2.5. Исходное дерево системных процессов

Ключевые системные процессы, показанные на рис. 2.5, рассматриваются в следующих разделах. Также будет кратко показан порядок запуска процессов.

Процесс простоя системы

Первым в списке на рис. 2.5 показан процесс простоя системы — Idle. Как будет объяснено в главе 5, процессы идентифицируются по именам их образов. Но этот процесс (и процесс System) не запускает какой-нибудь настоящий образ пользовательского режима (поэтому в каталоге \Windows не существует «System Idle Process.exe»). Кроме того, разными утилитами имя для этого процесса показывается по-разному (из-за особенностей реализации). В табл. 2.6 перечислен ряд имен, даваемых процессу Idle (с идентификатором процесса 0). Процесс Idle более подробно рассматривается в главе 5.

Таблица 2.6. Имена для процесса с идентификатором 0 в разных утилитах

Утилита	Имя для процесса с идентификатором 0
Task Manager	System Idle Process
Process Status (Pstat.exe)	Idle Process
Process Explorer (Procexp.exe)	System Idle Process
Task List (Tasklist.exe)	System Idle Process
Tlist (Tlist.exe)	System Process

Теперь давайте посмотрим на системные потоки и на предназначение каждого из системных процессов, в которых запущены реальные образы.

Процесс System и системные потоки

Процесс System (с идентификатором 4) дает начало потоку особого рода, запускаемому только в режиме ядра: *системному потоку режима ядра*. У системных потоков есть все атрибуты и контекстные составляющие, присущие обычным потокам пользовательского режима (например, контекст оборудования, приоритет и т. д.), но их отличие состоит в том, что они запускаются только в исполняемом коде в режиме ядра, который загружен в системное пространство, будь то код Ntoskrnl.exe или код любого другого загруженного драйвера устройства. Кроме того, у системных потоков нет адресного пространства пользовательского процесса и поэтому им нужно выделять любую динамическую память из динамически распределяемой памяти операционной системы, например выгружаемый или невыгружаемый пул.

Системные потоки создаются функцией `PsCreateSystemThread` (см. WDK), которая может быть вызвана только из режима ядра. Windows, а также различные драйверы устройств создают системные потоки при инициализации системы для выполнения операций, которым требуется контекст потока: например, для выдачи запросов на ввод-вывод и ожидания ответной реакции, или для ожидания реакции других объектов, или для опроса какого-нибудь устройства. Например, диспетчер памяти использует системные потоки для реализации таких функций, как запись измененных страниц в страничные файлы или в отображенные файлы, свопинг процессов в память и из памяти и т. д. Ядро создает системный поток под названием *диспетчер настройки баланса* (balance set manager), который активизируется каждую секунду для возможной инициации событий, связанных с планированием и управлением памятью. Системные потоки используются также диспетчером кэша для реализации опережающего чтения и отложенной записи при операциях ввода вывода. Драйвер файлового сервера (Srv2.sys) использует системные потоки для ответа на сетевые запросы ввода-вывода применительно к файловым данным, находящимся на общих для сети разделах диска. Для опроса устройства системные потоки есть даже у драйвера дисковода гибких дисков. (Опрос в данном случае более эффективен, поскольку привод гибких дисков, управляемый с помощью прерываний, расходует больше системных ресурсов.) Дополнительная информация о конкретных системных потоках включена в главы, в которых описывается тот или иной компонент.

По умолчанию системные потоки принадлежат процессу System, но драйвер устройства может создать системный поток в любом процессе. Например, драйвер устройства подсистемы Windows (Win32k.sys) создает системный поток внутри драйвера дисплея Canonical Display Driver (Cdd.dll), части процесса подсистемы Windows (Csrss.exe), чтобы он мог иметь упрощенный доступ к данным этого процесса, находящимся в адресном пространстве пользовательского режима.

При поиске неисправностей или анализе системы полезно сопоставить выполнение отдельных системных потоков с драйвером или даже с подпрограммой, содержащей код. Например, на сильно загруженном файловом сервере процесс System будет, скорее всего, потреблять существенную долю времени центрального процессора. Но для определения, какой именно драйвер устройства или компонент операционной системы запущен, будет недостаточно знания о том, что при запуске процесса System запускается «некий системный поток».

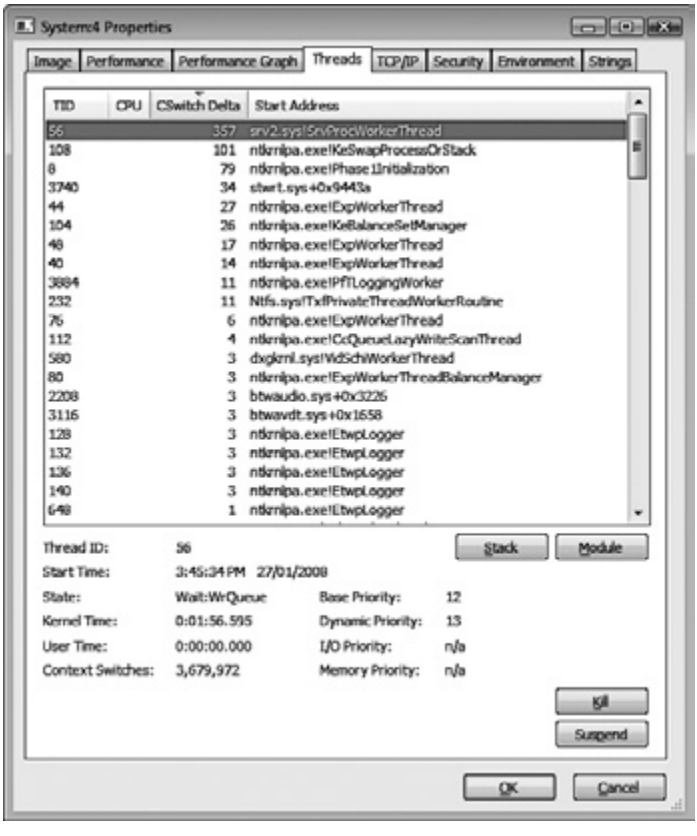
Поэтому, если потоки запущены в процессе System, сначала нужно определить, какой из них запущен (например, с помощью монитора производительности — Performance Monitor). При обнаружении запущенного потока (или потоков) посмотрите, в каком драйвере начал выполняться системный поток (что, по крайней мере, подскажет вам, какой именно драйвер, скорее всего, создал поток), или изучите стек вызовов (или, как минимум, текущий адрес) данного потока, что покажет, где поток выполняется в данное время.

Оба этих приема показаны в следующих экспериментах.

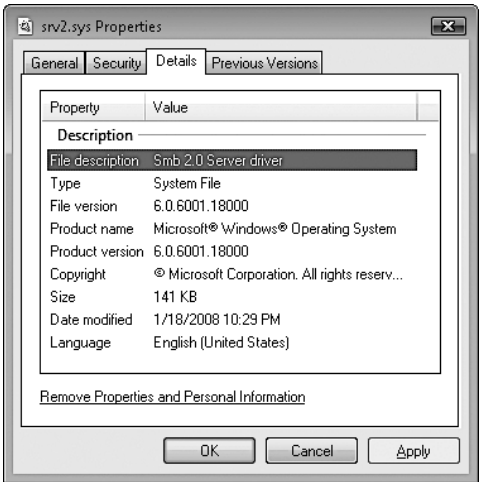
ЭКСПЕРИМЕНТ: ОТОБРАЖЕНИЕ СИСТЕМНОГО ПОТОКА НА ДРАЙВЕР УСТРОЙСТВА

В данном эксперименте будет показано, как установить соответствие активности центрального процессора с процессом System и с его системным потоком, который вызвал эту активность (и с драйвером, в который попадает этот поток). Это важно, потому что, когда запущен процесс System, нужно разобраться с его потоками, чтобы понять, что происходит. Для данного эксперимента мы сгенерируем активность системного потока путем генерации активности файлового сервера на вашей машине. (Драйвер файлового сервера, Srv2.sys, создает системные потоки для обработки входящих запросов файлового ввода-вывода. Дополнительная информация, касающаяся данного компонента, дана в главе 7.)

1. Откройте окно командной строки.
2. Выведите список каталогов всего вашего диска C, используя для доступа к нему сетевой путь. Например, если имя вашего компьютера `COMPUTER1`, наберите команду `dir \\computer1\c$ /s` (ключ `/s` приведет к выводу всех подкаталогов).
3. Запустите средство Process Explorer и дважды щелкните на строке процесса System.
4. Щелкните на вкладке Threads (Потоки).
5. Отсортируйте таблицу по столбцу CSwitch Delta (изменения в переключениях контекста). Вы должны увидеть один или несколько потоков, запущенных в Srv2.sys, таких как, например, следующие.



Если вы видите запущенный системный поток и не уверены, к какому драйверу он относится, щелкните на кнопке Module (Модуль), которая позволит извлечь свойства файла. Щелчок на кнопке Module (Модуль) при выделенном как на предыдущем рисунке потоком в Srv2.sys, приведет к отображению следующей информации.



Диспетчер сеанса (Smss)

Диспетчер сеанса (%SystemRoot%\System32\Smss.exe) является первым процессом пользовательского режима, созданным в системе. Этот процесс создается системным потоком режима ядра, выполняющим финальную фазу инициализации исполняющей системы и ядра.

При запуске Smss осуществляется проверка наличия его первого экземпляра (ведущего Smss) или своего собственного экземпляра, запущенного ведущим Smss для создания сеанса. (При наличии аргументов командной строки это будет последний экземпляр.) Путем создания нескольких экземпляров себя самого во время загрузки и создания сеанса Служб терминалов Smss может одновременно создать несколько сеансов (максимум четыре текущих сеанса, четыре параллельных сеанса, плюс еще один для каждого дополнительного центрального процессора, кроме первого). Эта возможность повышает производительность при входе в систему на системах Служб терминалов при одновременном подключении сразу нескольких пользователей. Когда завершится инициализация сеанса, копия Smss завершает свою работу. В результате остается активным только исходный процесс Smss.exe.

Ведущий Smss выполняет следующие единовременные этапы инициализации:

1. Помечает процесс и исходный поток как критический. (Если процесс или поток, помеченный как критический, по каким-то причинам завершается, происходит аварийное завершение работы Windows. Дополнительная информация дана в главе 5.)
2. Повышает базовый приоритет процесса до 11.
3. Если система поддерживает горячее добавление процессора, включает автоматическое обновление процессорного сходства, чтобы при добавлении нового процессора новые сеансы могли воспользоваться новыми процессорами. (Дополнительная информация о динамическом добавлении процессоров дана в главе 5.)
4. Создает поименованные каналы (pipes) и почтовые слоты (mailslots), используемые для связи между Smss, Csrss и Lsm (рассматриваемом далее).
5. Создает ALPC-порт для получения команд.
6. Создает общесистемные переменные среды окружения, определяемые в разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Environment.
7. Создает символические ссылки для устройств, определенных в разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\DOS Devices в каталоге \Global?? в пространстве имен диспетчера объектов.
8. Создает в пространстве имен диспетчера объектов корневой каталог \Sessions.
9. Запускает программы, указанные в разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\BootExecute. (По умолчанию там указывается программа Autochk.exe, осуществляющая проверку диска.)
10. Проводит отложенный перенос файлов, указанный в разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\PendingFileRenameOperations.

11. Инициализирует файл (файлы) подкачки.
12. Инициализирует всю остальную часть реестра (разделы HKLM Software, SAM и Security).
13. Запускает программы, указанные в разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\SetupExecute.
14. Открывает известные DLL-библиотеки (HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs) и отображает их как постоянные разделы (отображаемые файлы).
15. Создает поток, отвечающий за обработку запросов на создание сеанса.
16. Создает Smss для инициализации сеанса 0 (неинтерактивного сеанса).
17. Создает Smss для инициализации сеанса 1 (интерактивного сеанса).

Когда эти этапы будут выполнены, Smss переходит в режим постоянного ожидания дескриптора экземпляра Csrss.exe с нулевым сеансом. Поскольку Csrss имеет метку критического процесса, если происходит выход из Csrss, это ожидание из-за аварийного завершения работы системы никогда не заканчивается.

Экземпляр Smss, запускающий сеанс, выполняет следующие действия:

1. Вызывает функцию `NtSetSystemInformation` с запросом на создание структуры данных сеанса режима ядра. Та, в свою очередь, вызывает внутреннюю функцию диспетчера памяти `MmSessionCreate`, настраивающую виртуальное адресное пространство сеанса, которое будет содержать нерезидентный пул сеанса и структуру данных данного сеанса, выделяемые той частью подсистемы Windows, которая работает в режиме ядра (Win32k.sys), и другими драйверами устройств, относящимися к пространству сеанса.
2. Создает для сеанса процесс (процессы) подсистемы (по умолчанию Csrss.exe подсистемы Windows).
3. Создает экземпляр Winlogon (интерактивные сеансы) или Wininit (для сеанса 0). Дополнительная информация об этих двух процессах будет дана далее.

Затем происходит выход из этого промежуточного процесса Smss (после чего остаются процессы подсистемы, и в качестве процесса, не имеющего своего родительского процесса, остается процесс Winlogon или Wininit).

Процесс инициализации Windows (Wininit.exe)

Процесс Wininit.exe выполняет следующие функции инициализации системы:

- помечает сам себя как критический, чтобы при постоянных выходах из него и загрузке системы в режиме отладки он выходил в отладчик (если этого не случится, произойдет аварийное завершение работы системы);
- инициализирует инфраструктуру планирования пользовательского режима;
- создает папку %windir%\temp;
- создает станцию окна (Winsta0) и два рабочих стола (Winlogon и Default) для процессов, запускаемых в сеансе 0;
- создает поток Services.exe (Диспетчер управления службами — Service Control Manager или SCM);

- запускает Lsass.exe (Local Security Authentication Subsystem Server — Сервер проверки подлинности локальной системы безопасности);
- запускает Lsm.exe (Диспетчер локальных сеансов);
- входит в режим бесконечного ожидания завершения работы системы.

Диспетчер управления службами (SCM)

Вспомним, что ранее в данной главе под «службой» в Windows понимался либо серверный процесс, либо драйвер устройства. В этом разделе службы рассматриваются в качестве процессов пользовательского режима. Службы похожи на «процессы-демоны» в UNIX или на «обособленные процессы» VMS тем, что они могут быть настроены на автоматический запуск при загрузке системы, не требуя при этом интерактивного входа в систему. Они также могут быть запущены вручную (например, путем запуска средства администрирования Службы или путем вызова Windows-функции `StartService`). Обычно службы не взаимодействуют с пользователями, вошедшими в систему, хотя есть особые условия, открывающие такую возможность (см. главу 4).

Диспетчер управления службами является специальным системным процессом, запустившим образ %SystemRoot%\System32\Services.exe, отвечающим за запуск и остановку процессов служб, а также за взаимодействие с ними. Программы служб фактически являются Windows-образами, которые вызывают специальные Windows-функции для взаимодействия с диспетчером управления службами, чтобы выполнить такие действия, как регистрация успешного запуска службы, ответы на запросы о ее состоянии, или приостановки или полной остановки службы. Службы определены в реестре в разделе HKLM\SYSTEM\CurrentControlSet\Services.

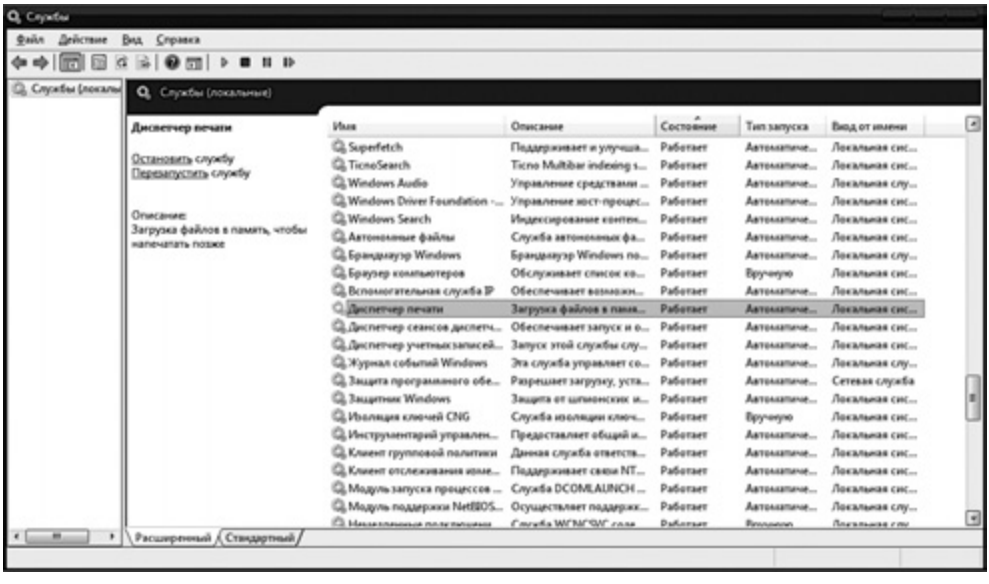
Следует иметь в виду, что у служб три имени: имя процесса, который виден запущенным в системе, внутреннее имя в реестре и имя, показываемое в средстве администрирования Службы. Отображаемое имя есть не у всех служб. Если у службы нет отображаемого имени, то показывается ее внутреннее имя. В Windows службы могут также иметь поле описания, дающее более глубокое представление о том, чем занимается та или иная служба.

Чтобы сопоставить процесс службы со службой, содержащейся в этом процессе, используется команда `tlist /s` или команда `tasklist /svc`. Следует заметить, что между процессами служб и запущенными службами однозначное соответствие бывает не всегда, потому что некоторые службы используют процесс совместно с другими службами. Код типа, имеющийся в реестре, показывает, запущена ли служба в своем собственный процессе или делит процесс с остальными службами образа.

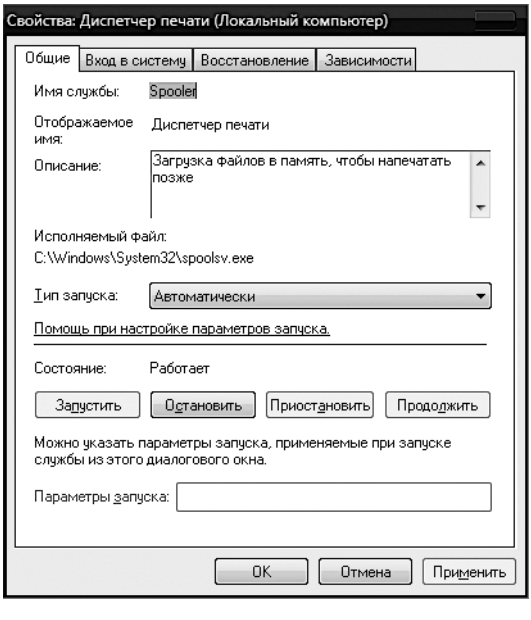
Некоторые компоненты Windows реализованы в виде служб. К ним относятся Диспетчер печати, Журнал событий, Планировщик заданий и различные сетевые компоненты. Более подробные сведения о службах даны в главе 4.

ЭКСПЕРИМЕНТ: ВЫВОД СПИСКА УСТАНОВЛЕННЫХ СЛУЖБ

Для вывода списка установленных служб выберите в окне Панель управления (Control Panel) пункт Администрирование (Administrative Tools), а затем выберите пункт Службы (Services). В результате должна быть выведена информация, похожая на следующую:



Чтобы увидеть детализированные свойства службы, щелкните на имени службы правой кнопкой мыши и выберите пункт Свойства (Properties). Например, на следующем рисунке показаны свойства службы под названием Диспетчер печати (чье имя выделено на предыдущем рисунке).



Обратите внимание на то, что в поле Исполняемый файл показана программа, содержащая данную службу. Следует помнить, что некоторые службы используют процесс совместно с другими службами, поэтому однозначное сопоставление службы и процесса получается не всегда. ■

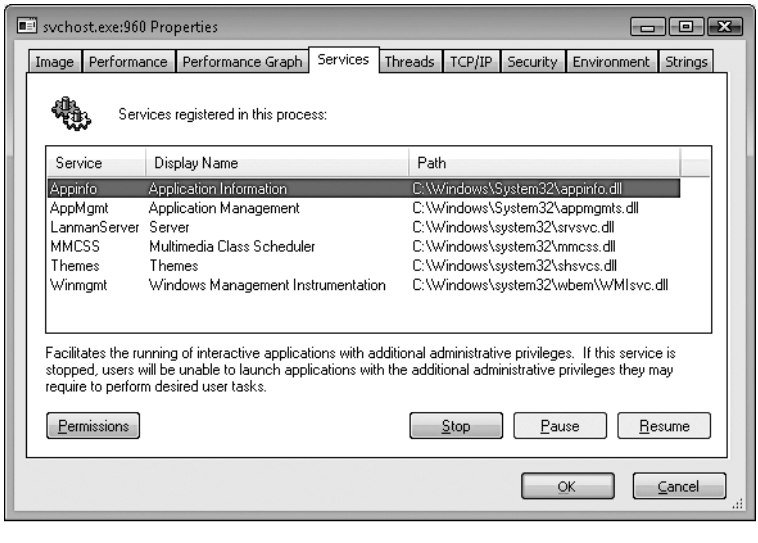
Диспетчер локальных сеансов (Lsm.exe)

Диспетчер локальных сеансов (Lsm.exe) управляет на локальной машине состоянием сеансов службы терминалов. Он отправляет через ALPC-порт SmSsWinStationApiPort запросы к Smss на запуск новых сеансов (например, на создание процессов Csrss и Winlogon), как при выборе пользователем в Explorer пункта Сменить пользователя (Switch User). Lsm также поддерживает связь с Winlogon и Csrss (используя RPC локальной системы). Он информирует Csrss о таких событиях, как подключение, отключение, завершение и рассылка системного сообщения. Он получает уведомление от Winlogon о следующих событиях:

- Вход и выход.
- Запуск и остановка оболочки.
- Подключение к сеансу.
- Отключение от сеанса.
- Установка или снятие блокировки рабочего стола.

ЭКСПЕРИМЕНТ: ПРОСМОТР ПОДРОБНОСТЕЙ СЛУЖБЫ В ЕЕ ПРОЦЕССАХ

Средство Process Explorer выделяет хост-процессы одной или нескольких служб. Свойство выделения можно настроить, выбрав в меню Options (Настройки) пункт Configure Colors (Цветовые настройки). Если дважды щелкнуть на имени хост-процесса одной или нескольких служб, во вкладке Services (Службы) можно увидеть перечень служб в процессе, имя параметра реестра, в котором определена служба, отображаемое имя, которое видит администратор, текст описания службы (если таковой имеется) и для служб Svchost путь к DLL-библиотеке, реализующей службу. Например, перечень служб в процессе Svchost.exe, запущенном под учетной записью System, выглядит следующим образом.



Winlogon, LogonUI и Userinit

Процесс входа в Windows (Winlogon, %SystemRoot%\System32\Winlogon.exe) обрабатывает интерактивные пользовательские входы в систему и выходы из нее. Winlogon получает уведомление о запросе пользователя на вход в систему, когда с помощью определенной комбинации клавиш вводится безопасная последовательность привлечения внимания — secure attention sequence (SAS). В качестве SAS в Windows используется комбинация Ctrl+Alt+Delete. Причиной применения SAS является защита пользователей от программ перехвата паролей, имитирующих процесс входа в систему (эта клавиатурная последовательность не может быть перехвачена приложением, работающем в пользовательском режиме).

Аспекты идентификации и аутентификации процесса входа в систему реализованы через DLL-библиотеки, называемые поставщиками учетных данных (credential providers). Стандартные поставщики учетных данных Windows реализуют интерфейсы аутентификации, используемые в Windows по умолчанию: паролей и смарт-карт. Но разработчики могут предоставить своих собственных поставщиков учетных данных для реализации вместо стандартного для Windows метода имя пользователя-пароль других механизмов идентификации и аутентификации (например, на основе распознавания голоса или биометрического устройства вроде сканера отпечатков пальцев). Поскольку Winlogon является критическим системным процессом, от которого зависит работа системы, поставщики учетных данных и пользовательский интерфейс для вывода диалогового окна входа в систему запускаются внутри дочернего процесса Winlogon под названием LogonUI. Когда Winlogon обнаруживает SAS, он запускает этот процесс, который инициализирует поставщиков учетных данных. Как только пользователь вводит свои учетные данные или отказывается от интерфейса входа в систему, процесс LogonUI завершается.

Кроме того, Winlogon может загрузить дополнительные DLL-библиотеки сетевых поставщиков (network providers), необходимые для дополнительной аутентификации. Эта возможность позволяет нескольким сетевым поставщикам за один раз во время обычного входа в систему собрать информацию, касающуюся идентификации и аутентификации.

Как только будут зарегистрированы имя пользователя и пароль, они будут отправлены для аутентификации процессу сервера проверки подлинности локальной системы безопасности (%SystemRoot%\System32\Lsass.exe, см. главу 6) to be authenticated. LSASS вызывает соответствующий пакет аутентификации (реализованный в виде DLL-библиотеки) для выполнения фактической проверки — например, для проверки соответствия пароля тому, что сохранено в Active Directory или в SAM (части реестра, в которой содержатся определения локальных пользователей и групп).

После успешной аутентификации LSASS вызывает функцию в мониторе безопасности (например, `NtCreateToken`) для генерации объекта маркера доступа, содержащего профиль безопасности пользователя. Если используется управление учетными записями пользователей (User Account Control, UAC) и входящий в систему пользователь относится к группе администраторов или имеет права администратора, LSASS создаст вторую, ограниченную версию маркера. Затем этот маркер доступа используется Winlogon для создания исходного процесса

(процессов) в пользовательском сеансе. Исходный процесс (процессы) хранится в параметре реестра Userinit, который находится в разделе HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon. (По умолчанию это Userinit.exe, но в списке может быть более одного образа.)

Userinit выполняет действия по инициализации пользовательской среды окружения (например, запускает сценарий входа в систему и применяет групповую политику), а затем смотрит в реестр на значение параметра Shell (в разделе Winlogon) и создает процесс для запуска определяемой системой оболочки (по умолчанию Explorer.exe). Затем Userinit завершается. Именно поэтому для процесса Explorer.exe родительский процесс не показывается — его родительский процесс завершился, и, как объяснялось в главе 1, средство tlist выравнивает имена процессов, чьи родительские процессы не работают по левому краю. (По другому это выглядит так, что процесс Explorer является внуком процесса Winlogon.)

Winlogon является активным не только во время входа пользователя в систему и выхода из нее, но также и при перехвате SAS с клавиатуры. Например, если войдя в систему нажать комбинацию клавиш Ctrl+Alt+Delete, появится экран безопасности Windows, предоставляющий возможности для выхода из системы, запуска диспетчера задач, блокировки компьютера и т. д. Этим взаимодействием занимаются процессы Winlogon и LogonUI.

Подробности проверки подлинности рассмотрены в главе 6. Подробности вызываемых функций, устанавливающих связь с LSASS (функций, чьи имена начинаются с префикса Lsa), даны в документации по Windows SDK.

Заключение

В данной главе мы ознакомились с общей системной архитектурой Windows. Мы изучили основные компоненты Windows и их взаимосвязи. В следующей главе более подробно будут рассмотрены основные системные механизмы, на которых построены эти компоненты, такие как диспетчер объектов и синхронизация.

Глава 3. Системные механизмы

Операционная система Windows предоставляет ряд основных механизмов, используемых такими компонентами режима ядра, как исполняющая система, ядро и драйверы устройств. В данной главе рассматриваются следующие системные механизмы и описывается порядок их использования:

- диспетчеризация системных прерываний, включая прерывания, отложенные вызовы процедур (DPC), асинхронные вызовы процедур (APC), диспетчеризация исключений и диспетчеризация системных служб;
- диспетчер объектов исполняющей системы;
- синхронизация, включая спин-блокировки, объекты диспетчера ядра, порядок реализации ожиданий, а также примитивы синхронизации, относящиеся к пользовательскому режиму и избегающие переходов в режим ядра (в отличие от обычных объектов диспетчера);
- системные рабочие потоки;
- различные механизмы, например глобальные флаги Windows;
- усовершенствованные вызовы локальных процедур (ALPC);
- трассировка событий ядра (Kernel event tracing);
- Wow64;
- отладка в пользовательском режиме;
- загрузчик образов (image loader);
- гипервизор (Hyper-V);
- диспетчер транзакций ядра (Kernel Transaction Manager, KTM);
- защита ядра от исправлений (Kernel Patch Protection, KPP);
- обеспечение целостности кода.

Диспетчеризация системных прерываний

Прерывания и исключения являются условиями операционной системы, отвлекающими процессор на выполнение кода, находящегося за пределами нормального потока управления. Они могут быть обнаружены либо аппаратными, либо программными средствами. Термин *системное прерывание* относится к механизму процессора, предназначенному для захвата выполняемого потока при возникновении исключения или прерывания и для передачи управления в определенное место в операционной системе. В Windows процессор передает управление *обработчику системного прерывания*, который является функцией, характерной для того или иного прерывания или исключения. На рис. 3.1 проиллюстрированы некоторые условия активации обработчиков системных прерываний.

Ядро различает прерывания и исключения следующим образом. *Прерывание* является асинхронным событием (которое может произойти в любое время), не связанным с текущей задачей процессора. Прерывания генерируются, главным образом, устройствами ввода-вывода, процессорными часами или таймерами, и они могут быть разрешены (включены) или запрещены (выключены). *Исключение*, напротив, является синхронным условием, которое обычно возникает в резуль-

тате выполнения конкретной инструкции. (Аварийные завершения работы, например, из-за машинного сбоя, обычно не связаны с выполнением инструкции.) Повторение исключений может быть вызвано повторным запуском программы с теми же данными и при тех же условиях. В качестве примеров исключений можно привести нарушения доступа к памяти, определенные инструкции отладки и ошибки деления на нуль. Ядро также считает исключениями вызовы системных служб (хотя технически они являются системными прерываниями).

Рис. 3.1. Диспетчеризация системных прерываний

Исключения и прерывания могут генерироваться как оборудованием, так и программами. Например, причиной исключения, связанного с ошибкой шины, является оборудование, а исключение, связанное с делением на нуль, является результатом программной ошибки. Точно так же прерывание может генерироваться устройством ввода-вывода, или же программное прерывание может быть выдано самим ядром (прерывания APC или DPC будут рассмотрены в этой главе).

При генерации аппаратного исключения или прерывания процессор записывает довольно большой объем информации о состоянии машины в стек ядра того потока, который был прерван, для возвращения к нужной точке потока управления и продолжения выполнения, как будто ничего не случилось. Если поток выполнялся в пользовательском режиме, Windows переключается на стек потоков режима ядра. Затем Windows создает в стеке ядра *фрейм системного прерывания* прерванного потока, в котором она сохраняет состояние выполнения потока. Фрейм системного прерывания является подмножеством полного контекста потока, и его определение можно увидеть, набрав в отладчике ядра команду `dt nt!_ktrap_frame` (см. главу 5). Ядро обрабатывает программные прерывания либо как часть обработки аппаратных прерываний, либо одновременно с ними, когда поток вызывает функции ядра, относящиеся к программному прерыванию.

В большинстве случаев ядро устанавливает внешние функции обработки системных прерываний, которые выполняют общие задачи их обработки до и после передачи управления другим функциям, выставившим системное прерывание. Например, если ситуация была вызвана прерыванием от какого-нибудь устройства, находящийся в ядре обработчик аппаратных системных прерываний передает управление *процедуре обработки прерывания* (Interrupt service routine, ISR), которую драйвер устройства предоставил для устройства, вызвавшего прерывание. Если ситуация создалась из-за вызова системной службы, общий обработчик системных прерываний, связанных с системными службами, передает управление конкретной функции системной службы в исполняющей системе. Ядро также устанавливает обработчики системных прерываний для неожиданных или необрабатываемых им прерываний. Эти обработчики системных прерываний обычно выполняют системную функцию KeBugCheckEx, останавливающую компьютер, когда ядро обнаруживает проблемное или некорректное поведение, которое, если его не предотвратить, может привести к повреждению данных. Более подробно прерывания, исключения и диспетчеризация системных служб рассматриваются в следующих разделах.

Диспетчеризация прерываний

Аппаратно генерируемые прерывания обычно исходят от устройств ввода-вывода, которые должны уведомить процессор о том, что они нуждаются в обслуживании. Устройства, управляемые прерываниями, позволяют операционной системе использовать процессор максимально эффективно, совмещая основную обработку данных с операциями ввода-вывода. Поток запускает передачу ввода-вывода в адрес устройства или от него, а затем может выполнять другую полезную работу, пока устройство не завершит передачу. Когда устройство завершит работу, оно прерывает процессор на свое обслуживание. Как правило, прерываниями управляются устройства указания координат, принтеры, клавиатуры, дисковые приводы и сетевые карты.

Прерывания могут также генерироваться системными программами. Например, ядро может выдать программное прерывание для инициирования диспетчера потока и для асинхронного проникновения в выполнение потока.

Ядро может также запретить прерывания, но делается это крайне редко, только в критических ситуациях, например во время программирования контроллера прерываний или диспетчеризации исключений. Для ответа на прерывания, исходящие от устройств, ядро устанавливает обработчики системных прерываний. Эти обработчики передают управление либо внешней процедуре (ISR), обрабатывающей прерывание, либо внутренней процедуре ядра, реагирующей на прерывание. Для обслуживания прерываний от устройств драйверы устройств предоставляют ISR-процедуры, а ядро предоставляет процедуры, обрабатывающие другие типы прерываний.

В следующих разделах вы узнаете, как оборудование уведомляет процессор о прерываниях, исходящих от устройств, о типах прерываний, поддерживаемых ядром, о том, как драйверы устройств взаимодействуют с ядром (в том, что касается обработки прерываний), о том, какие программные прерывания распознаются ядром, а также об объектах ядра, используемых для их реализации.

Обработка аппаратных прерываний

На аппаратных платформах, поддерживаемых Windows, внешние прерывания ввода-вывода поступают на одну из линий контроллера прерываний. В свою очередь, контроллер прерывает работу процессора по отдельной линии. Как только работа процессора будет прервана, этот процессор требует от контроллера запрос прерывания (Interrupt request, IRQ). Контроллер прерываний превращает IRQ в номер прерывания, использует этот номер в качестве индекса в структуре под названием *таблица диспетчеризации прерываний* (Interrupt dispatch table, IDT) и передает управление соответствующей процедуре обработки прерывания. Во время загрузки системы Windows заполняет IDT указателями на процедуры ядра, обрабатывающими каждое прерывание и исключение.

Windows отображает аппаратные IRQ-запросы на номера прерываний в IDT и также использует IDT для настройки обработчиков системных прерываний для исключений. Например, в системах x86 и x64 номер исключения для ошибки отсутствия страницы (исключения, которое возникает, когда поток пытается обратиться к странице виртуальной памяти, которая не определена или отсутствует) имеет значение 0xe (14). Таким образом, запись 0xe в IDT указывает на системный обработчик ошибки обращения к странице. Хотя архитектура, поддерживаемая Windows, допускает до 256 записей в IDT-таблице, количество IRQ-запросов, поддерживаемых на конкретной машине, определяется конструкцией используемого контроллера прерываний.

ЭКСПЕРИМЕНТ: ПРОСМОТР IDT

Содержимое IDT, включая информацию о том, какие обработчики системных прерываний назначены операционной системой Windows прерываниям (включая исключения и IRQ-запросы), можно посмотреть, используя команду отладки ядра !idt. Команда !idt без ключей показывает упрощенный вывод, который включает только зарегистрированные аппаратные прерывания (и на 64-разрядных машинах обработчики системных прерываний процессора).

В следующем примере показано, как выглядит вывод команды !idt:

```
lkd> !idt
Dumping IDT:
00:  fffff80001a7ec40 nt!KiDivideErrorFault
01:  fffff80001a7ed40 nt!KiDebugTrapOrFault
02:  fffff80001a7ef00 nt!KiNmiInterrupt  Stack = 0xFFFFF80001865000
03:  fffff80001a7f280 nt!KiBreakpointTrap
04:  fffff80001a7f380 nt!KiOverflowTrap
05:  fffff80001a7f480 nt!KiBoundFault
06:  fffff80001a7f580 nt!KiInvalidOpcodeFault
07:  fffff80001a7f7c0 nt!KiNpxNotAvailableFault
08:  fffff80001a7f880 nt!KiDoubleFaultAbort  Stack = 0xFFFFF80001863000
09:  fffff80001a7f940 nt!KiNpxSegmentOverrunAbort
0a:  fffff80001a7fa00 nt!KiInvalidTssFault
0b:  fffff80001a7fac0 nt!KiSegmentNotPresentFault
0c:  fffff80001a7fc00 nt!KiStackFault
0d:  fffff80001a7fd40 nt!KiGeneralProtectionFault
```

продолжение ⇩

```
0e:  fffff80001a7fe80 nt!KiPageFault
10:  fffff80001a80240 nt!KiFloatingErrorFault
11:  fffff80001a803c0 nt!KiAlignmentFault
12:  fffff80001a804c0 nt!KiMcheckAbort  Stack = 0xFFFFF80001867000
13:  fffff80001a80840 nt!KiXmmException
1f:  fffff80001a5ec10 nt!KiApcInterrupt
2c:  fffff80001a80a00 nt!KiRaiseAssertion
2d:  fffff80001a80b00 nt!KiDebugServiceTrap
2f:  fffff80001acd590 nt!KiDpcInterrupt
37:  fffff8000201c090 hal!PicSpuriousService37 (KINTERRUPT fffff8000201c000)
3f:  fffff8000201c130 hal!PicSpuriousService37 (KINTERRUPT fffff8000201c0a0)
51:  fffffa80045babd0 dxgkrnl!DpiFdoLineInterruptRoutine (KINTERRUPT fffffa80045bab40)
52:  fffffa80029f1390 USBPORT!USBPORT_InterruptService (KINTERRUPT fffffa80029f1300)
62:  fffffa80029f15d0 USBPORT!USBPORT_InterruptService (KINTERRUPT fffffa80029f1540)
           USBPORT!USBPORT_InterruptService (KINTERRUPT fffffa80029f1240)
72:  fffffa80029f1e10 ataport!IdePortInterrupt (KINTERRUPT fffffa80029f1d80)
81:  fffffa80045bae10 i8042prt!I8042KeyboardInterruptService (KINTERRUPT
     fffffa80045bad80)
82:  fffffa80029f1ed0 ataport!IdePortInterrupt (KINTERRUPT fffffa80029f1e40)
90:  fffffa80045bad50 Vid+0x7918 (KINTERRUPT fffffa80045bacc0)
91:  fffffa80045baed0 i8042prt!I8042MouseInterruptService (KINTERRUPT fffffa80045bae40)
a0:  fffffa80045bac90 vmbus!XPartPncIsr (KINTERRUPT fffffa80045bac00)
a2:  fffffa80029f1210 sdbus!SdbusInterrupt (KINTERRUPT fffffa80029f1180)
           rimmpx64+0x9FFC (KINTERRUPT fffffa80029f10c0)
           rimspx64+0x7A14 (KINTERRUPT fffffa80029f1000)
           rixdpx64+0x9C50 (KINTERRUPT fffffa80045baf00)
a3:  fffffa80029f1510 USBPORT!USBPORT_InterruptService (KINTERRUPT fffffa80029f1480)
           HDAudBus!HdaController::Isr (KINTERRUPT fffffa80029f1c00)
a8:  fffffa80029f1bd0 NDIS!ndisMiniportMessageIsr (KINTERRUPT fffffa80029f1b40)
a9:  fffffa80029f1b10 NDIS!ndisMiniportMessageIsr (KINTERRUPT fffffa80029f1a80)
aa:  fffffa80029f1a50 NDIS!ndisMiniportMessageIsr (KINTERRUPT fffffa80029f19c0)
ab:  fffffa80029f1990 NDIS!ndisMiniportMessageIsr (KINTERRUPT fffffa80029f1900)
ac:  fffffa80029f18d0 NDIS!ndisMiniportMessageIsr (KINTERRUPT fffffa80029f1840)
ad:  fffffa80029f1810 NDIS!ndisMiniportMessageIsr (KINTERRUPT fffffa80029f1780)
ae:  fffffa80029f1750 NDIS!ndisMiniportMessageIsr (KINTERRUPT fffffa80029f16c0)
af:  fffffa80029f1690 NDIS!ndisMiniportMessageIsr (KINTERRUPT fffffa80029f1600)
b0:  fffffa80029f1d50 NDIS!ndisMiniportMessageIsr (KINTERRUPT fffffa80029f1cc0)
b1:  fffffa80029f1f90 ACPI!ACPIInterruptServiceRoutine (KINTERRUPT fffffa80029f1f00)
b3:  fffffa80029f1450 USBPORT!USBPORT_InterruptService (KINTERRUPT fffffa80029f13c0)
c1:  fffff8000201c3b0 hal!HalpBroadcastCallService (KINTERRUPT fffff8000201c320)
d1:  fffff8000201c450 hal!HalpHpetClockInterrupt (KINTERRUPT fffff8000201c3c0)
d2:  fffff8000201c4f0 hal!HalpHpetRolloverInterrupt (KINTERRUPT fffff8000201c460)
df:  fffff8000201c310 hal!HalpApicRebootService (KINTERRUPT fffff8000201c280)
e1:  fffff80001a8e1f0 nt!KiIpiInterrupt
e2:  fffff8000201c270 hal!HalpDeferredRecoveryService (KINTERRUPT fffff8000201c1e0)
e3:  fffff8000201c1d0 hal!HalpLocalApicErrorService (KINTERRUPT fffff8000201c140)
fd:  fffff8000201c590 hal!HalpProfileInterrupt (KINTERRUPT fffff8000201c500)
fe:  fffff8000201c630 hal!HalpPerfInterrupt (KINTERRUPT fffff8000201c5a0)
```

На системе, предоставившей вывод для данного эксперимента, ISR-процедура клавиатуры в драйвере устройства (I8042prt.sys) назначена прерыванию с номером 0x81. Можно также увидеть, что как ранее было объяснено, прерывание 0xe соответствует системной функции KiPageFault. ■

Каждый процессор имеет отдельную IDT-таблицу, поэтому другие процессоры, если нужно, могут запускать другие ISR-процедуры. Например, в мультипроцессорной системе прерывание от часов получает каждый процессор, но только один процессор в ответ на это прерывание обновляет показания системных часов. Тем не менее все процессоры используют прерывание для замера кванта времени потока и для инициации перепланирования по истечении этого кванта.

Кроме того, некоторые настройки системы могут требовать обработку определенных прерываний от устройств конкретным процессором.

Контроллер прерываний x86

В большинстве систем x86 используется либо программируемый контроллер прерываний (Programmable Interrupt Controller, PIC) i8259A, либо один из вариантов усовершенствованного программируемого контроллера прерываний (Advanced Programmable Interrupt Controller, APIC) i82489. Современные контроллеры комплектуются контроллером APIC. Стандарт PIC возник с появлением IBM PC. PIC i8259A работает только с однопроцессорными системами и имеет только 8 линий прерываний. Но в архитектуре IBM PC определена возможность использования дополнительного второго PIC-контроллера, называемого ведомым, чьи прерывания мультиплексируются в одну из линий прерываний ведущего PIC-контроллера. Тем самым общее число прерываний доводится до 15 (7 на ведущем и 8 на ведомом контроллере, мультиплексируемые на восьмую линию прерывания ведущего контроллера). С мультипроцессорными системами работают APIC- и SAPIC-контроллеры (Streamlined Advanced Programmable Interrupt Controllers — модернизированные усовершенствованные программируемые контроллеры прерываний — мы вскоре рассмотрим), имеющие 256 линий прерываний. Intel и ряд других компаний определили мультипроцессорную спецификацию — Multiprocessor Specification (MP Specification), конструкторский стандарт для мультипроцессорных систем x86, в основу которого заложено применение APIC. Для обеспечения совместимости с однопроцессорными операционными системами и кодом начальной загрузки, который запускает мультипроцессорную систему в однопроцессорном режиме, APIC-контроллеры поддерживают режим совместимости с 15 прерываниями и доставки прерываний только основному процессору. Архитектура APIC-контроллера изображена на рис. 3.2.

Фактически APIC-контроллер состоит из нескольких компонентов: APIC ввода-вывода, получающего прерывания от устройств, локальных APIC-контроллеров, получающих прерывания от APIC ввода-вывода на шине и прерывающих работу того центрального процессора, с которым они связаны, и i8259A-совместимого контроллера прерываний, который переводит APIC-ввод в сигналы, эквивалентные PIC-контроллеру. Поскольку в системе может быть несколько APIC-контроллеров ввода-вывода, у материнских плат обычно есть часть основных логических устройств, расположенных между ними и процессорами. Эти логические устройства отвечают за реализацию алгоритмов процедур прерываний, которые наряду с балансировкой нагрузки аппаратных прерываний между

процессорами и попыткой получить преимущества от локализации, доставляют прерывания от устройств тому же самому процессору, которому только что было направлено предыдущее прерывание того же типа. Обычные программы могут перепрограммировать APIC-контроллеры ввода-вывода, применив фиксированный алгоритм маршрутизации, обходящий логику набора микросхем. Windows делает это путем программирования APIC-контроллеров в режиме маршрутизации «прерывания одного процессора в следующем наборе».

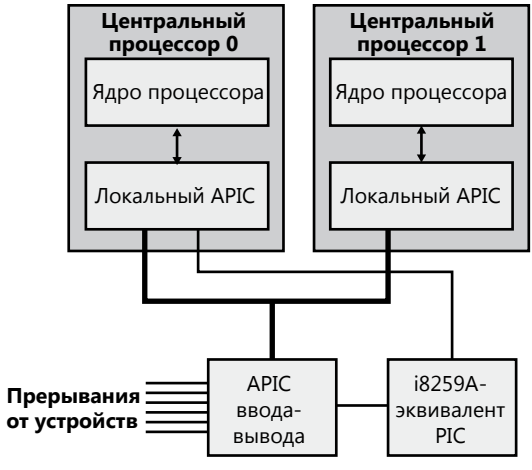


Рис. 3.2. APIC-архитектура x86

Контроллеры прерываний x64

Поскольку архитектура x64 совместима с операционными системами x86, системы x64 должны предоставлять такие же контроллеры прерываний, что и системы x86. Но существенная разница состоит в том, что x64-версии не будут запускаться на системах, которые не имеют APIC, поскольку для управления прерываниями они используют APIC.

Контроллеры прерываний IA64

Архитектура IA64 зависит от модернизированного усовершенствованного программируемого контроллера Streamlined Advanced Programmable Interrupt Controller (SAPIC), который является результатом развития контроллера APIC. Даже если во встроенной программе присутствует балансировка и маршрутизация, Windows ими не пользуется, вместо этого она назначает прерывания процессорам статически, по кругу.

ЭКСПЕРИМЕНТ: ПРОСМОТР PIC И APIC

Конфигурацию PIC на однопроцессорной системе и текущего локального APIC на мультипроцессорной системе можно просмотреть соответственно с помощью команд отладчика ядра !pic и !apic. Вывод команды !pic на однопроцессорной системе имеет следующий вид[1].

[1] Следует учесть, что команда !pic в системе, использующей APIC HAL, не работает.

```
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  .  .  .  .
```

Далее следует вывод команды !apic, запущенной на системе, использующей APIC HAL. Учтите, что при локальной отладке ядра эта команда показывает APIC, связанный с текущим процессором, — иначе говоря, с тем процессором, на котором был запущен поток отладчика при вводе команды. При просмотре аварийного дампа или удаленной системы для переключения на тот процессор, чей локальный APIC нужно просмотреть, можно использовать команду ~ (тильда) с указанием после нее номера процессора.

```
lkd> !apic
Apic @ fffe0000  ID:0 (50014)  LogDesc:01000000  DestFmt:ffffffff  TPR 20
TimeCnt: 00000000clk  SpurVec:3f  FaultVec:e3  error:0
Ipi Cmd: 01000000'0000002f  Vec:2F  FixedDel    Ph:01000000      edg high
Timer..: 00000000'000300fd  Vec:FD  FixedDel    Dest=Self      edg high      m
Linti0.: 00000000'0001003f  Vec:3F  FixedDel    Dest=Self      edg high      m
Linti1.: 00000000'000004ff  Vec:FF  NMI         Dest=Self      edg high
TMR: 51-52, 62, A3, B1, B3
IRR:
ISR::
```

Различные числа, следующие за метками Vec, показывают вектор в IDT, связанный с заданной командой. Например, в данном выводе прерывание номер 0xFD связано с APIC Timer, а прерывание номер 0xE3 управляет ошибками APIC. Поскольку этот эксперимент был запущен на той же самой машине, что и ранее проводившийся эксперимент !idt, можно заметить, что 0xFD является прерыванием профилирования HAL (которое использует таймер для профилирования интервалов), а 0xe3, как и ожидалось, является обработчиком ошибок локального APIC HAL.

Следующий вывод является результатом выполнения команды !ioapic, показывающей конфигурацию APIC-контроллеров ввода-вывода, компонентов контроллера прерываний, подключенного к устройствам:

```
lkd> !ioapic
IoApic @ FEC00000  ID:0 (51)  Arb:A951

Inti00.: 0000a951'0000a951  Vec:51  LowestDl  Lg:0000a951      lvl low     ■
```

Уровни запросов программных прерываний (IRQL)

Хотя контроллеры прерываний устанавливают приоритетность прерываний, Windows устанавливает свою собственную схему приоритетности прерываний, известную как *уровни запросов прерываний* (IRQL). В ядре IRQL-уровни представлены в виде номеров от 0 до 31 на системах x86 и в виде номеров от 0 до 15 на системах x64 и IA64, где более высоким номерам соответствуют прерывания с более высоким приоритетом. Хотя ядро определяет для программных прерываний стандартный набор IRQL-уровней, HAL отображает номера аппаратных пре-

рываний на IRQL-уровни. На рис. 3.3 показаны IRQL-уровни для архитектуры x86, а на рис. 3.4 показаны IRQL-уровни для архитектур x64 и IA64.

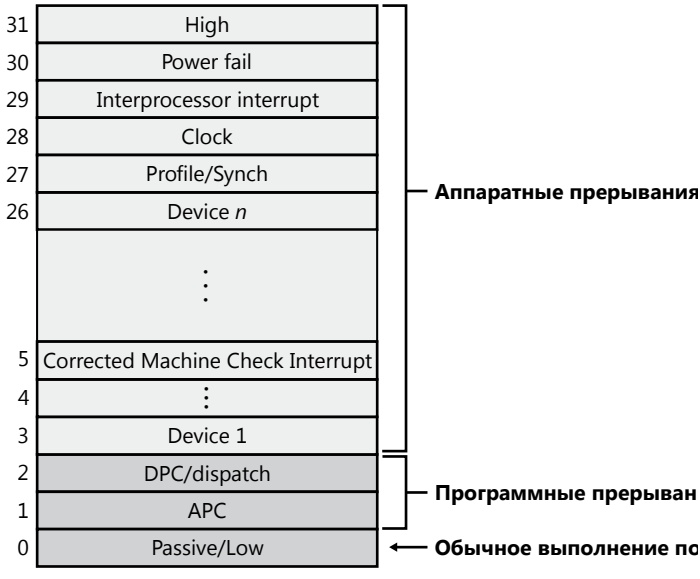


Рис. 3.3. Уровни запросов прерываний (IRQL) для архитектуры x86

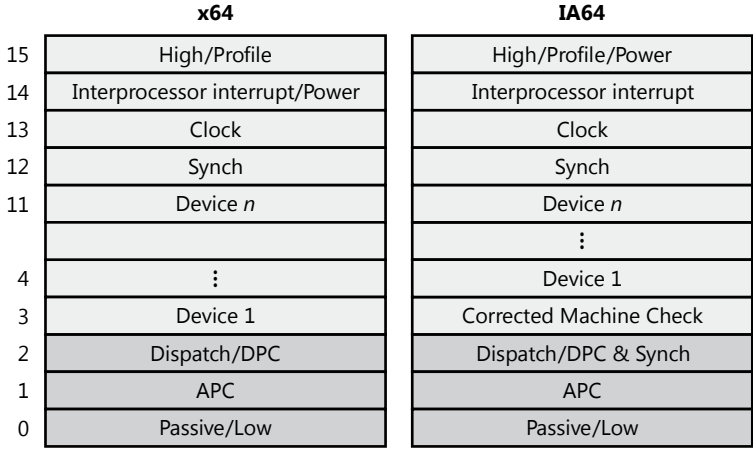


Рис. 3.4. Уровни запросов прерываний (IRQL) для архитектур x64 и IA64

Прерывания обслуживаются в порядке их приоритета, и прерывания с более высоким уровнем приоритета получают преимущество в обслуживании. При возникновении прерывания с высоким уровнем процессор сохраняет состояние прерванного потока и запускает связанный с прерыванием диспетчер системных прерываний. Тот, в свою очередь, поднимает IRQL и вызывает процедуру обработки прерывания. После выполнения этой процедуры диспетчер прерываний понижает IRQL-уровень процессора до значения, на котором он был до

возникновения прерывания, а затем загружает сохраненное состояние машины. Прерванный поток продолжает выполнение с того места, в котором оно было прервано. Когда ядро понижает IRQL, могут реализоваться те прерывания с более низким уровнем приоритета, которые были замаскированы. Если так и происходит, ядро повторяет процесс для обработки новых прерываний.

Уровни приоритетов IRQL имеют совершенно другое значение, чем приоритеты, используемые при планировании потоков (которые рассматриваются в главе 5). Приоритет планирования является атрибутом потока, а IRQL является атрибутом источника прерывания, такого как клавиатура или мышь. Кроме того, у каждого процессора есть установка IRQL, которая меняется при выполнении кода операционной системы.

Установка IRQL каждого процессора определяет, какие прерывания данный процессор может получать. IRQL-уровни также используются для синхронизации доступа к структуре данных режима ядра. Как только запускается поток режима ядра, он повышает или понижает IRQL процессора либо напрямую, путем вызова функций KeRaiseIrql и KeLowerIrql, либо, что случается чаще, опосредованно, через вызовы функций, которые запрашивают объекты ядра, используемые для синхронизации. Как показано на рис. 3.5, прерывания, поступающие от источника с IRQL, превышающим текущий уровень, прерывают работу процессора, а прерывания, поступающие от источников с IRQL-уровнями равными или ниже текущего уровня, *маскируются* до тех пор, пока выполняющийся поток не понизит IRQL.

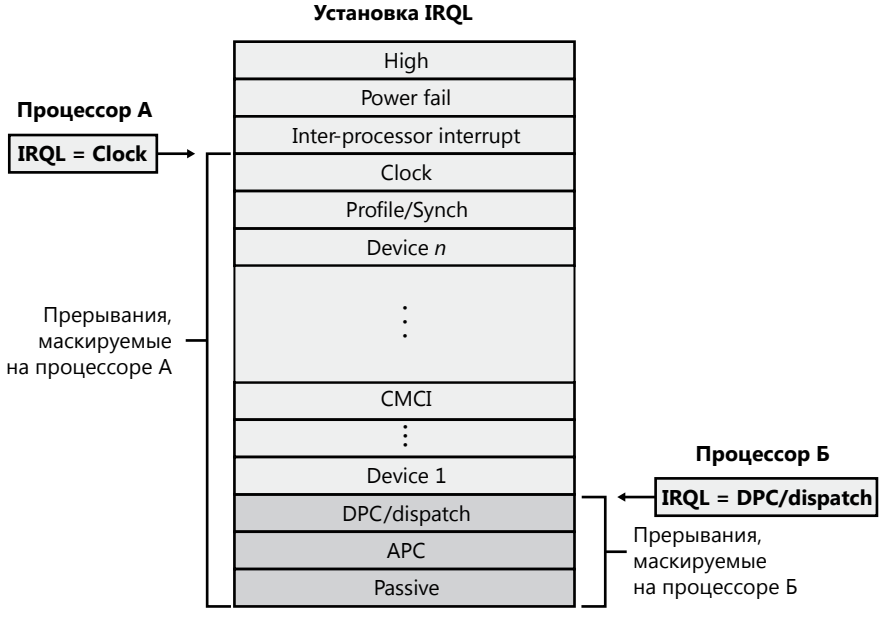


Рис. 3.5. Маскирование прерываний

Поскольку обращение к PIC является относительно медленной операцией, HAL-механизмы, требующие доступа к шине ввода-вывода для изменения IRQL-уровней (например, для PIC и 32-разрядных систем усовершенствованного интерфейса управления конфигурированием и энергопотреблением — Advanced Configuration

and Power Interface, ACPI), реализуют оптимизацию производительности, которая называется «ленивой IRQL» (lazy IRQL) и позволяет избежать обращений к PIC. При повышении IRQL HAL отмечает для себя новый IRQL, не изменяя маски прерывания. Если же после этого произойдет прерывание с более низким уровнем, HAL устанавливает маску прерывания с настройками, соответствующими первому прерыванию, и не замораживает прерывание с более низким уровнем (сохраняя его тем самым в отложенном состоянии) до тех пор, пока IRQL не будет понижен. Таким образом, если прерываний с более низким уровнем при повышении IRQL не происходит, модифицировать PIC HAL-механизмам не требуется.

ПРИМЕЧАНИЕ

Исключения из правила блокировки прерываний равного или более низкого уровня при повышении IRQL касаются прерываний APC-уровня. Если поток повышает IRQL до уровня APC, а затем его выполнение подвергается перепланированию из-за прерывания dispatch/DPC-уровня, система может передать прерывание APC-уровня заново спланированному потоку. Таким образом, уровень APC может считаться IRQL, локальным для потока, а не для всего процессора.

Поток режима ядра повышает или понижает IRQL того процессора, на котором он запущен, в зависимости от того, что он пытается сделать. Например, когда возникает прерывание, обработчик системных прерываний (или, возможно, процессор) повышает IRQL процессора до IRQL, установленного для источника прерывания. Это повышение маскирует все прерывания с таким же и более низким IRQL (только на этом процессоре), гарантируя, что процессор, обслуживающий прерывание, не захватывается прерываниями на том же или на более низком уровне. Замаскированные прерывания либо обрабатываются другим процессором, либо придерживаются до тех пор, пока IRQL не упадет. Поэтому все компоненты системы, включая ядро и драйверы устройств, стараются держать IRQL на пассивном уровне (который иногда называют низким уровнем). Это сделано для своевременной реакции драйверов устройств на аппаратные прерывания при условии, что IRQL не держится неоправданно высоким в течение продолжительных периодов времени.

ЭКСПЕРИМЕНТ: ПРОСМОТР IRQL

Просмотреть сохраненный для процессора IRQL можно с помощью команды отладчика !irql. Сохраненный IRQL представляет собой IRQL на момент, непосредственно предшествующий проникновению в отладчик, повышающему IRQL до статического, ничего не значащего значения:

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

Учтите, что значение IRQL сохраняется в двух местах. Первое место, в котором представлен текущий IRQL, — это область управления процессором (processor control region, PCR), а второе — его расширение, блок управления областью процессора (processor region control block, PRCB), который содержит сохраненный IRQL в поле DebuggerSaveIrql. PCR и PRCB содержат информацию о состоянии каждого процессора в системе, в том числе текущий IRQL, указатель на аппаратную IDT-таблицу, сведения о текущем потоке

и следующем, выбранном для запуска потоке. Ядро и HAL используют эту информацию для выполнения действий, ориентированных на конкретную архитектуру и конкретную машину. Части структур PCR и PRCB открыто определены в заголовочном файле Ntddk.h, относящемся к инструментарию Windows Driver Kit (WDK).

Содержимое PCR текущего процессора можно просмотреть с помощью отладчика ядра, используя команду !pcr. Для просмотра PCR конкретного процессора нужно после команды добавить номер процессора, отделив его от команды пробелом:

```
lkd> !pcr 0
KPCR for Processor 0 at  fffff80001bfad00:
    Major 1 Minor 1
    NtTib.ExceptionList: fffff80001853000
        NtTib.StackBase: fffff80001854080
       NtTib.StackLimit: 000000000026ea28
     NtTib.SubSystemTib: fffff80001bfad00
          NtTib.Version: 0000000001bfae80
      NtTib.UserPointer: fffff80001bfb4f0
          NtTib.SelfTib: 000007fffffdb000
                SelfPcr: 0000000000000000
                   Prcb: fffff80001bfae80
                   Irql: 0000000000000000
                    IRR: 0000000000000000
                    IDR: 0000000000000000
          InterruptMode: 0000000000000000
                    IDT: 0000000000000000
                    GDT: 0000000000000000
                    TSS: 0000000000000000
          CurrentThread: fffff80001c08c40
             NextThread: 0000000000000000
             IdleThread: fffff80001c08c40
              DpcQueue:
```

■

Поскольку изменение IRQL процессора существенно влияет на работу системы, это изменение должно вноситься только в режиме ядра — потоки пользовательского режима внести это изменение не могут. Это означает, что IRQL процессора при выполнении кода в пользовательском режиме всегда находится на пассивном уровне. Уровень IRQL может быть повышен только при выполнении кода в режиме ядра.

У каждого уровня прерывания есть своя конкретная цель. Например, ядро выдает *межпроцессорное прерывание* (Interprocessor interrupt, IPI), чтобы запросить выполнение действия на другом процессоре, например диспетчеризацию конкретного потока для выполнения или обновления кэш-памяти его буфера быстрого преобразования адреса (Translation look-aside buffer, TLB). Через определенные периоды времени системные часы генерируют прерывание, а ядро реагирует на него, обновляя значение часов и измеряя время выполнения потока. Если аппаратная платформа поддерживает двое часов, ядро добавляет еще один уровень прерывания от системных часов для измерения производительности.

HAL предоставляет сразу несколько уровней прерываний для использования устройствами, управляемыми с помощью прерываний. Конкретное количество этих уровней зависит от процессора и конфигурации системы. Ядро использует программные прерывания (рассматриваемые далее) для инициации планирования потоков и для асинхронного вмешательства в выполнение потока.

Отображение прерываний на IRQL-уровни

IRQL-уровни нельзя отождествлять с запросами на прерывание (IRQ), определенными контроллерами прерываний, — в архитектурах, на которых работает Windows, концепция IRQL на аппаратном уровне не реализована. Как же тогда Windows определяет, какой IRQL следует назначить прерыванию? Ответ можно найти в HAL. В Windows присутствие устройств на шине (PCI, USB и т. д.) и прерываний, которые могут быть назначены этим устройствам, определяется таким типом драйвера устройства, который называется *драйвером шины*. Драйвер шины сообщает эту информацию диспетчеру устройств Plug and Play, который, с учетом допустимых назначения прерываний для всех остальных устройств, решает, какое прерывание будет назначено каждому устройству. Затем им вызывается арбитр прерываний Plug and Play, который отображает прерывания на IRQL-уровни. (На системах, не имеющих ACPI, используется корневой арбитр, а на ACPI-совместимых системах ACPI HAL имеет свой собственный арбитр.)

Для разных HAL, включенных в Windows, используются разные алгоритмы назначения. На ACPI-системах (включая x86, x64 и IA64), HAL вычисляет IRQL для заданного прерывания путем деления вектора прерывания, назначенного IRQ на 16. Что же касается выбора вектора прерывания для IRQ, это зависит от типа контроллера прерывания, имеющегося в системе. На современных APIC-системах это число генерируется в циклическом режиме, поэтому какого-либо способа вычислить IRQ на основе вектора прерывания или IRQL не существует. Но в следующем эксперименте (см. с. 117) будет показано, как отладчик может запросить эту информацию у арбитра прерываний.

Предопределенные IRQL-уровни

Давайте приглядимся к использованию к предопределенным IRQL, показанным на рис. 3.4, начиная с самого высокого уровня:

- Ядро использует высокий уровень (*high*), только когда оно останавливает систему в функции KeBugCheckEx и маскирует все прерывания.
- Уровень сбоя электропитания (*power fail*) впервые упоминается в исходных документах о конструкции Windows NT. Он определяет поведение системного кода при сбое электропитания, но этот IRQL никогда не использовался.
- Уровень межпроцессорного прерывания (*interprocessor interrupt*) используется для запроса выполнения действия на другом процессоре, например для обновления кэш-памяти процессорного TLB, завершения работы системы или при аварийном отказе системы.
- Уровень часов (*clock*) используется для системных часов. С его помощью ядро отслеживает время суток, а также измеряет и выделяет потокам время центрального процессора.

- Системные часы реального времени (или другой источник, например, локальный таймер APIC) используют при включении профилирования ядра (при задействовании механизма измерения производительности) уровень *profile*. При активном состоянии профилирования ядра его обработчик системного прерывания записывает адрес кода, который выполнялся в тот момент, когда возникло прерывание. Со временем создается таблица выборки адресов, которую можно извлечь и проанализировать с помощью инструментов. Для настройки и просмотра сгенерированной при профилировании статистики можно использовать такое средство, как Kernrate, входящее в состав Windows Driver Kit (WDK). Дополнительная информация об использовании данного средства дана в эксперименте, проводимом с Kernrate.
- IRQL синхронизации (*synchronization*) используется внутри операционной системы кодом диспетчера и планировщика для защиты доступа к глобальному потоку планирования и к коду ожидания-синхронизации. Обычно этот уровень определяется как самый высокий сразу же после IRQL-уровней устройств.
- IRQL-уровни устройств (*device*) используются для определения приоритетов прерываний, связанных с устройствами. Порядок отображения уровней прерываний на IRQL-уровни показан в предыдущем разделе.
- Уровень скорректированной машинной проверки (*corrected machine check*) используется для оповещения операционной системы после серьезной, но скорректированной аппаратной неблагоприятной ситуации или ошибки, о которой через интерфейс Machine Check Error (MCE) сообщил центральный процессор или встроенное программное обеспечение.
- Уровень отложенного вызова процедуры и диспетчеризации (*DPC/dispatch*) и уровень асинхронного вызова процедуры (*APC*) относятся к программным прерываниям, генерируемым ядром и драйверами устройств. Более подробно вызовы DPC и APC будут рассмотрены позже.
- Самый низкий пассивный IRQL-уровень (*passive*) на самом деле вообще не является настоящим уровнем прерываний; он соответствует обычному выполнению потока и возможности возникновения всех прерываний.

ЭКСПЕРИМЕНТ: ИСПОЛЬЗОВАНИЕ СРЕДСТВА ПРОФИЛИРОВАНИЯ ЯДРА (KERNRATE) ДЛЯ ЗАМЕРА ПРОИЗВОДИТЕЛЬНОСТИ

Средство Kernel Profiler (Kernrate) можно использовать для включения таймера профилирования системы, сбора примеров кода, выполняемых при запусках таймера, и вывода итоговой информации о распределении процессорного времени между файлами образов задач и функциями. Оно может быть использовано для отслеживания времени центрального процессора, потребляемого отдельными процессами и (или) времени, проведенном в режиме ядра независимо от процессов (например, затраченном на процедуры обслуживания прерывания). Профилирование ядра применяется при необходимости разобраться в том, на что система тратит время.

В своей самой простой форме Kernrate предоставляет выборку затрат времени на каждый модуль ядра (например, на Ntoskrnl, драйверы и т. д.). Например, после установки Windows Driver Kit попробуйте выполнить следующие действия.

1. Откройте окно командной строки.
2. Наберите cd C:\WinDDK\7600.16385.1\tools\other (путь к вашей установке WDK для Windows 7/Server 2008R2).
3. Наберите dir. Вы увидите каталоги для каждой платформы.
4. Запустите образ, соответствующий вашей платформе (без аргументов или ключей). Например, для системы x86 используется образ i386\kernrate.exe.
5. Во время работы Kernrate выполните на системе какие-нибудь другие действия. Например, запустите Windows Media Player и проиграйте какую-нибудь музыку, запустите игру, широко использующую графику, или сделайте что-нибудь в сети вроде вывода содержимого каталога на удаленном общем сетевом ресурсе.
6. Нажмите Ctrl+C для остановки Kernrate. Это заставит Kernrate показать статистику за период выборки.

В следующем выводе выборки из Kernrate показано, что был запущен проигрыватель Windows Media Player, воспроизводящий фильм с диска:

```
C:\WinDDK\7600.16385.1\tools\Other\i386>kernrate.exe
 /==============================\
<         KERNRATE LOG           >
 \==============================/
Date: 2011/03/09   Time: 16:44:24
Machine Name: TEST-LAPTOP
Number of Processors: 2
PROCESSOR_ARCHITECTURE: x86
PROCESSOR_LEVEL: 6
PROCESSOR_REVISION: 0f06
Physical Memory: 3310 MB
Pagefile Total: 7285 MB
Virtual Total: 2047 MB
PageFile1: \??\C:\pagefile.sys, 4100MB
OS Version: 6.1 Build 7601 Service-Pack: 1.0
WinDir: C:\Windows
Kernrate Executable Location: C:\WINDDK\7600.16385.1\TOOLS\OTHER\I386
Kernrate User-Specified Command Line:
kernrate.exe
Kernel Profile (PID = 0): Source= Time,
Using Kernrate Default Rate of 25000 events/hit
Starting to collect profile data
***> Press ctrl-c to finish collecting profile data
===> Finished Collecting Data, Starting to Process Results
------------Overall Summary:--------------
P0     K 0:00:00.000 ( 0.0%)  U 0:00:00.234 ( 4.7%)  I 0:00:04.789 (95.3%)
DPC 0:00:00.000 ( 0.0%)  Interrupt 0:00:00.000 ( 0.0%)
       Interrupts= 9254, Interrupt Rate= 1842/sec.
P1     K 0:00:00.031 ( 0.6%)  U 0:00:00.140 ( 2.8%)  I 0:00:04.851 (96.6%)
DPC 0:00:00.000 ( 0.0%)  Interrupt 0:00:00.000 ( 0.0%)
       Interrupts= 7051, Interrupt Rate= 1404/sec.
TOTAL  K 0:00:00.031 ( 0.3%)  U 0:00:00.374 ( 3.7%)  I 0:00:09.640 (96.0%)
```

```
DPC 0:00:00.000 ( 0.0%)  Interrupt 0:00:00.000 ( 0.0%)
       Total Interrupts= 16305, Total Interrupt Rate= 3246/sec.
Total Profile Time = 5023 msec
                                BytesStart       BytesStop          BytesDiff.
  Available Physical Memory ,    1716359168,      1716195328,         -163840
  Available Pagefile(s)     ,    5973733376,      5972783104,         -950272
  Available Virtual         ,    2122145792,      2122145792,               0
 Available Extended Virtual ,             0,               0,               0
  Committed Memory Bytes    ,    1665404928,      1666355200,          950272
 Non Paged Pool Usage Bytes ,      66211840,        66211840,               0
  Paged Pool Usage Bytes    ,     189083648,       189087744,            4096
  Paged Pool Available Bytes,     150593536,       150593536,               0
  Free System PTEs          ,         37322,           37322,               0
                                   Total          Avg. Rate
    Context Switches     ,         30152,          6003/sec.
    System Calls         ,        110807,         22059/sec.
    Page Faults          ,           226,          45/sec.
    I/O Read Operations  ,           730,          145/sec.
    I/O Write Operations ,          1038,          207/sec.
    I/O Other Operations ,           858,          171/sec.
    I/O Read Bytes       ,       2013850,          2759/ I/O
    I/O Write Bytes      ,         28212,          27/ I/O
    I/O Other Bytes      ,         19902,          23/ I/O
-----------------------------
Results for Kernel Mode:
-----------------------------
OutputResults: KernelModuleCount = 167
Percentage in the following table is based on the Total Hits for the Kernel
Time   3814 hits, 25000 events per hit --------
Module                                   Hits       msec  %Total  Events/Sec
NTKRNLPA                                 3768       5036    98 %    18705321
NVLDDMKM                                   12       5036     0 %       59571
HAL                                        12       5036     0 %       59571
WIN32K                                     10       5037     0 %       49632
DXGKRNL                                     9       5036     0 %       44678
NETW4V32                                    2       5036     0 %        9928
FLTMGR                                      1       5036     0 %        4964
================================= END OF RUN =================================
============================== NORMAL END OF RUN ==============================
```

Общие итоги показывают, что система провела 0,3 % времени в режиме ядра, 3,7 % в пользовательском режиме, 96,0 % в простое, 0,0 % на уровне DPC и 0,0 % на уровне прерывания. Наиболее востребованным модулем был Ntkrnlpa.exe, ядро для машин с поддержкой расширения физического адреса (Physical Address Extension, PAE) или NX. Вторым по востребованности модулем является nvlddmkm.sys, драйвер для видеокарты на машине, используемой для тестирования. В этом есть смысл, поскольку основная активность, проявляемая в системе, исходила от Windows Media Player, отправлявшего ввод-вывод видео на видеодрайвер.

Если у вас есть файлы символов, вы можете расширить отдельные модули и посмотреть затраченное время, разбитое по именам функций. Например, профилирование системы при быстром перетаскивании окна по экрану приведет к следующему выводу (который показан частично):

```
 C:\WinDDK\7600.16385.1\tools\Other\i386>kernrate.exe -z ntkrnlpa -z win32k
 /============================\
<         KERNRATE LOG           >
 \============================/
Date: 2011/03/09   Time: 16:49:56

Time   4191 hits, 25000 events per hit --------
Module                                  Hits       msec  %Total  Events/Sec
NTKRNLPA                                3623       5695    86 %    15904302
WIN32K                                   303       5696     7 %     1329880
INTELPPM                                 141       5696     3 %      618855
HAL                                       61       5695     1 %      267778
CDD                                       30       5696     0 %      131671
NVLDDMKM                                  13       5696     0 %       57057

--- Zoomed module WIN32K.SYS (Bucket size = 16 bytes, Rounding Down) --------
Module                                  Hits       msec  %Total  Events/Sec
BltLnkReadPat                             34       5696    10 %      149227
memmove                                   21       5696     6 %       92169
vSrcTranCopyS8D32                         17       5696     5 %       74613
memcpy                                    12       5696     3 %       52668
RGNOBJ::bMerge                            10       5696     3 %       43890
HANDLELOCK::vLockHandle                    8       5696     2 %       35112

--- Zoomed module NTKRNLPA.EXE (Bucket size = 16 bytes, Rounding Down) --------
Module                                  Hits       msec  %Total  Events/Sec
KiIdleLoop                              3288       5695    87 %    14433713
READ_REGISTER_USHORT                      95       5695     2 %      417032
READ_REGISTER_ULONG                       93       5695     2 %      408252
RtlFillMemoryUlong                        31       5695     0 %      136084
KiFastCallEntry                           18       5695     0 %       79016
```

Вторым по востребованности модулем был Win32k.sys, драйвер системы управления окнами. Также высокое место в списке было у видеодрайвера и у Cdd.dll, глобального видеодрайвера, используемого для темы рабочего стола Aero с 3D-ускорением. Эти результаты вполне понятны, поскольку основная активность системы заключалась в прорисовке экрана. Заметьте, что в расширенном отображении для Win32k.sys наиболее востребованными оказались функции, связанные с объединением, копированием и перемещением битов, основными GDI-операциями для прорисовки окна, перетаскиваемого по экрану. ■

Одним из важных ограничений, накладываемых на код, выполняемый на уровне DPC/dispatch или выше, является то, что он не может ждать объект, если это требует от планировщика выбрать для выполнения другой поток, что является недопустимой операцией, поскольку планировщик при планировании по-

токов зависит от программных прерываний DPC-уровня. Другим ограничением является то, что на IRQL-уровне DPC/dispatch или выше может быть доступна только неперемещаемая область памяти.

Это правило фактически является побочным эффектом первого ограничения, поскольку попытка обращения к нерезидентной памяти приводит к ошибке обращения к странице. При возникновении такой ошибки диспетчер памяти инициирует дисковый ввод-вывод, а затем вынуждает ждать, пока драйвер файловой системы считает страницу с диска.

Это ожидание, в свою очередь, заставляет планировщик выполнить контекстное переключение (возможно, на поток простоя, idle, если запуска не ожидает ни один пользовательский поток), нарушая тем самым правило, по которому планировщик не может быть вызван, поскольку во время чтения диска код попрежнему выполняется на IRQL-уровне DPC/dispatch или выше. Еще одна проблема является следствием того факта, что завершение ввода-вывода происходит на уровне APC_LEVEL, следовательно, даже в тех случаях, когда ожидания не потребуется, ввод-вывод никогда не завершится, поскольку завершение APC не получит никакого шанса на выполнение.

При нарушении любого их этих двух ограничений происходит сбой системы с кодом аварийного завершения работы IRQL_NOT_LESS_OR_EQUAL (IRQL не является меньшим или равным) или DRIVER_IRQL_NOT_LESS_OR_EQUAL (IRQL драйвера не является меньшим или равным). Нарушение этих ограничений является распространенной ошибкой при создании драйверов устройств. В качестве средства, помогающего обнаружить ошибки подобного типа, можно использовать верификатор Windows Driver Verifier.

Объекты прерываний

Ядро предоставляет переносимый механизм — управляющий объект ядра, называемый *объектом прерывания*, который позволяет драйверам устройств регистрировать процедуры обработки прерываний (ISR) для своих устройств. Объект прерывания содержит всю информацию, необходимую ядру для того, чтобы связать ISR устройства с конкретным уровнем прерывания, включая адрес ISR, IRQL-уровень, на котором прерывается устройство, и запись в таблице диспетчеризации прерываний ядра (IDT), с которой нужно сопоставить ISR. При инициализации объекта прерывания в нем сохраняются несколько инструкций на языке ассемблера, так называемый *код диспетчеризации*, копируемый из шаблона обработки прерывания, KiInterruptTemplate. Этот код выполняется при возникновении прерывания.

Этот резидентный код объекта прерывания вызывает настоящий диспетчер прерывания, которым обычно является либо процедура KiInterruptDispatch, либо процедура KiChainedDispatch, передавая ему указатель на объект прерывания. KiInterruptDispatch является процедурой, используемой для векторов прерывания, для которых зарегистрирован только один объект прерывания. KiChainedDispatch является процедурой для векторов, совместно используемых несколькими объектами прерываний. Объект прерывания содержит информацию, которая нужна этой второй процедуре диспетчера для обнаружения и правильного вызова ISR-процедуры, предоставляемой драйвером устройства.

Объект прерывания также хранит в себе IRQL, связанный с прерыванием, чтобы процедура KiInterruptDispatch или KiChainedDispatch перед вызовом ISR смогла поднять IRQL на правильный уровень, а затем, после возвращения управления из ISR, соответствующим образом понизить IRQL. Этот процесс, выполняемый в два этапа, нужен потому, что при начальной диспетчеризации способов передачи указателя (или какого-нибудь другого аргумента на этот случай) объекту прерывания не существует, поскольку начальная диспетчеризация осуществляется оборудованием. На мультипроцессорной системе ядро выделяет и инициализирует объект прерывания для каждого центрального процессора, позволяя локальному APIC этого процессора принимать конкретное прерывание.

На Windows-системах x64 ядро оптимизирует диспетчер прерываний, используя специальные процедуры, сохраняющие циклы процессора, пропуская ненужные функции, например функцию KiInterruptDispatchNoLock, используемую для прерываний, не имеющих связанную с ними и управляемую ядром спин-блокировку (обычно используется драйверами, требующими синхронизации с их ISR-процедурами), и функцию KiInterruptDispatchNoEOI. Функция KiInterruptDispatchNoEOI используется для прерываний, запрограммировавших APIC в режим автоматического завершения прерывания — «Auto-End-of-Interrupt» (Auto-EOI), — поскольку контроллер прерываний пошлет сигнал EOI автоматически, ядру не нужно выполнять дополнительный код для самостоятельного осуществления EOI. И наконец, конкретно для прерывания профилирования производительности (performance/profiling) используется обработчик KiInterruptDispatchLBControl, поддерживающий регистр управления последним условным переходом Last Branch Control MSR, доступный на современных центральных процессорах. Этот регистр позволяет ядру отслеживать или сохранять инструкцию условного перехода при трассировке; при прерывании эта информация будет потеряна, поскольку она не сохраняется в контексте регистра обычного прерывания, поэтому для его сохранения должен быть добавлен специальный код. Эта функция используется, к примеру, прерываниями производительности и профилирования HAL, в то время как другие процедуры прерываний HAL пользуются «неблокируемым» кодом диспетчеризации, поскольку HAL не требует от ядра синхронизации с его ISR.

Еще один обработчик прерывания ядра KiFloatingDispatch используется для прерываний, требующих сохранения состояния в формате числа с плавающей точкой. В отличие от кода режима ядра, которому обычно не разрешается использовать операции с плавающей точкой (MMX, SSE, 3DNow!), поскольку относящиеся к ним регистры не будут сохраняться при переключениях контекста, ISR-процедуры могут нуждаться в использовании этих регистров (например, ISR видеокарты, выполняющей быструю операцию рисования). При подключении прерывания драйверы могут установить аргумент FloatingSave в TRUE, требуя от ядра использования процедуры диспетчеризации с числами с плавающей точкой, которая сохранит регистры, используемые для этих чисел (это существенно увеличивает время обработки прерывания). Следует заметить, что эта функция поддерживается только на 32-разрядных системах.

На рис. 3.6 показана типовая схема управления прерываниями, связанными с объектами прерываний.

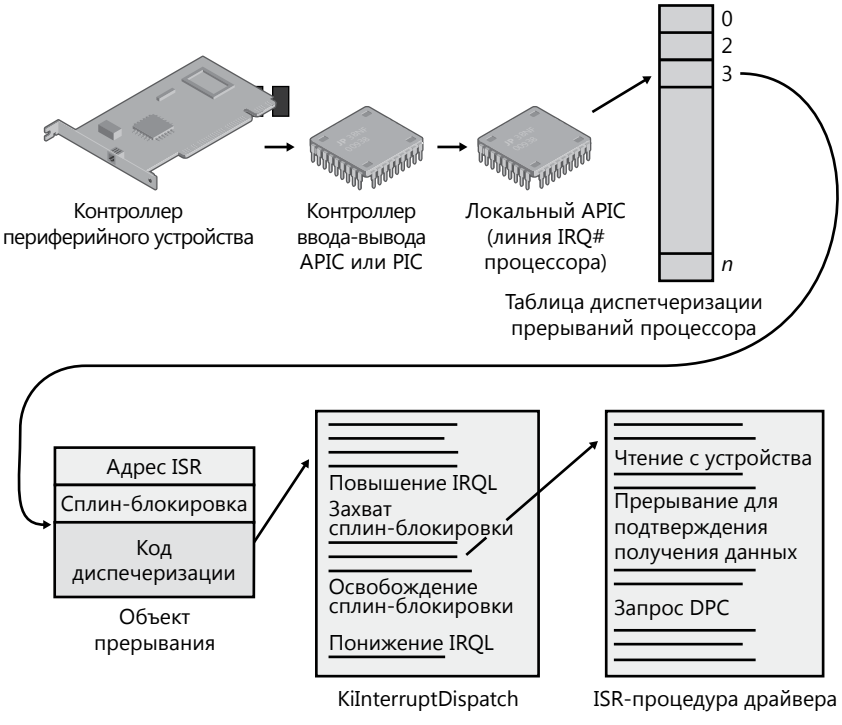


Рис. 3.6. Типовая схема управления прерываниями

ЭКСПЕРИМЕНТ: ИЗУЧЕНИЕ ВНУТРЕННЕГО УСТРОЙСТВА ПРЕРЫВАНИЙ

Используя отладчик ядра, можно просмотреть детали объекта прерывания, включая его IRQL, адрес ISR и специализированный код диспетчеризации прерывания. Сначала нужно запустить команду !idt и обнаружить запись, включающую ссылку на I8042KeyboardInterruptService, ISR-процедуру для PS2-клавиатуры:

```
81:    fffffa80045bae10 i8042prt!I8042KeyboardInterruptService (KINTERRUPT
fffffa80045bad80)
```

Для просмотра содержимого объекта прерывания, связанного с данным прерыванием, нужно запустить команду dt nt!_kinterrupt с адресом, следующим за KINTERRUPT:

```
lkd> dt nt!_KINTERRUPT fffffa80045bad80
   +0x000 Type             : 22
   +0x002 Size             : 160
   +0x008 InterruptListEntry : _LIST_ENTRY [ 0x00000000'00000000 - 0x0 ]
   +0x018 ServiceRoutine   : 0xfffff880'0356ca04     unsigned char
  i8042prt!I8042KeyboardInterruptService+0
   +0x020 MessageServiceRoutine : (null)
   +0x028 MessageIndex     : 0
   +0x030 ServiceContext   : 0xfffffa80'02c839f0
```

продолжение ⇩

```
   +0x038 SpinLock        : 0
   +0x040 TickCount       : 0
   +0x048 ActualLock      : 0xfffffa80'02c83b50  -> 0
   +0x050 DispatchAddress : 0xfffff800'01a7db90     void  nt!KiInterruptDispatch+0
   +0x058 Vector          : 0x81
   +0x05c Irql            : 0x8 ''
   +0x05d SynchronizeIrql : 0x9 ''
   +0x05e FloatingSave    : 0 ''
   +0x05f Connected       : 0x1 ''
   +0x060 Number          : 0
   +0x064 ShareVector     : 0 ''
   +0x065 Pad             : [3]  ""
   +0x068 Mode            : 1 ( Latched )
   +0x06c Polarity        : 0 ( InterruptPolarityUnknown )
   +0x070 ServiceCount    : 0
   +0x074 DispatchCount   : 0
   +0x078 Rsvd1           : 0
   +0x080 TrapFrame       : 0xfffff800'0185ab00 _KTRAP_FRAME
   +0x088 Reserved        : (null)
   +0x090 DispatchCode    : [4] 0x8d485550
```

В данном примере IRQL, присваиваемый Windows прерыванию, равен 8. Хотя прямого отображения вектора прерывания на IRQ нет, Windows отслеживает это преобразование при управлении ресурсами устройства через механизм так называемых *арбитров*. Для каждого типа ресурсов арбитр поддерживает связь между виртуальным использованием ресурсов (например, вектором прерывания) и физическими ресурсами (например, линией прерывания). Таким образом можно запросить либо корневой IRQ-арбитр (на системах без ACPI), либо ACPI IRQ-арбитр и получить это отображение. Для получения информации об ACPI IRQ-арбитре нужно воспользоваться командой !apciirqarb:

```
lkd> !acpiirqarb
Processor 0 (0, 0):
Device Object: 0000000000000000
Current IDT Allocation:
...
   0000000000000081 - 0000000000000081   D   fffffa80029b4c20  (i8042prt)
A:0000000000000000 IRQ:0
...
```

Если используется система без ACPI, можно воспользоваться командой !arbiter 4 (Цифра 4 заставит отладчик показать только IRQ-арбитры):

```
lkd> !arbiter 4
DEVNODE fffffa80027c6d90 (HTREE\ROOT\0)
  Interrupt Arbiter "RootIRQ" at fffff80001c82500
    Allocated ranges:
      0000000000000081 - 0000000000000081   Owner    fffffa80029b4c20 (i8042prt)
```

В обоих случаях вам будет предоставлен владелец вектора по типу объекта устройства. Затем вы можете воспользоваться командой !devobj, чтобы

получить в этом примере информацию об устройстве i8042prt (которое соответствует драйверу PS/2):

```
lkd> !devobj fffffa80029b4c20
Device object (fffffa80029b4c20) is for:
 00000061 \Driver\ACPI DriverObject fffffa8002888e70
Current Irp 00000000 RefCount 1 Type 00000032 Flags 00003040
Dacl fffff9a100096a41 DevExt fffffa800299f740 DevObjExt fffffa80029b4d70 DevNode
fffffa80029b54b0
```

Объект устройства связан с узлом устройства, хранящим все физические ресурсы этого устройства.

Теперь с помощью команды !devnode можно вывести дамп этих ресурсов и воспользоваться ключом 6, чтобы запросить информацию о ресурсах:

```
lkd> !devnode fffffa80029b54b0 6
DevNode 0xfffffa80029b54b0 for PDO 0xfffffa80029b4c20
  Parent 0xfffffa800299b390   Sibling 0xfffffa80029b5230   Child 0000000000
  InstancePath is "ACPI\PNP0303\4&17aa870d&0"
  ServiceName is "i8042prt"
...
  CmResourceList at 0xfffff8a00185bf40  Version 1.1  Interface 0xf  Bus #0
    Entry 0 - Port (0x1) Device Exclusive (0x1)
      Flags (0x11) - PORT_MEMORY PORT_IO 16_BIT_DECODE
      Range starts at 0x60 for 0x1 bytes
    Entry 1 - Port (0x1) Device Exclusive (0x1)
      Flags (0x11) - PORT_MEMORY PORT_IO 16_BIT_DECODE
      Range starts at 0x64 for 0x1 bytes
    Entry 2 - Port (0x1) Device Exclusive (0x1)
      Flags (0x11) - PORT_MEMORY PORT_IO 16_BIT_DECODE
      Range starts at 0x62 for 0x1 bytes
    Entry 3 - Port (0x1) Device Exclusive (0x1)
      Flags (0x11) - PORT_MEMORY PORT_IO 16_BIT_DECODE
      Range starts at 0x66 for 0x1 bytes
    Entry 4 - Interrupt (0x2) Device Exclusive (0x1)
      Flags (0x01) - LATCHED
      Level 0x1, Vector 0x1, Group 0, Affinity 0xffffffff
```

Узел устройства сообщает, что у данного устройства есть перечень ресурсов с четырьмя записями, одна из которых является записью прерывания, соответствующего IRQ 1. (Номера уровня и вектора представляют вектор IRQ, а не вектор прерывания.) IRQ 1 является традиционным номером PC/AT IRQ, связанным с клавиатурой PS/2, следовательно, это вполне ожидаемое значение. (У USB-клавиатуры будет другое прерывание.)

На ACPI-системах эту информацию можно получить более легким путем, прочитав расширенный вывод представленной ранее команды !acpiirqarb. В качестве части своего вывода эта команда показывает таблицу отображения IRQ на IDT:

```
Interrupt Controller (Inputs: 0x0-0x17  Dev: 0000000000000000):
     (00)Cur:IDT-a1 Ref-1 edg hi   Pos:IDT-00 Ref-0 edg hi
    (01)Cur:IDT-81 Ref-1 edg hi   Pos:IDT-00 Ref-0 edg hi
```

продолжение ⇩

```
(02)Cur:IDT-00 Ref-0 edg hi   Pos:IDT-00 Ref-0 edg hi
(03)Cur:IDT-00 Ref-0 edg hi   Pos:IDT-00 Ref-0 edg hi
(04)Cur:IDT-00 Ref-0 edg hi   Pos:IDT-00 Ref-0 edg hi
(05)Cur:IDT-00 Ref-0 edg hi   Pos:IDT-00 Ref-0 edg hi
(06)Cur:IDT-00 Ref-0 edg hi   Pos:IDT-00 Ref-0 edg hi
(07)Cur:IDT-00 Ref-0 edg hi   Pos:IDT-00 Ref-0 edg hi
(08)Cur:IDT-71 Ref-1 edg hi   Pos:IDT-00 Ref-0 edg hi
(09)Cur:IDT-b1 Ref-1 lev hi   Pos:IDT-00 Ref-0 edg hi
(0a)Cur:IDT-00 Ref-0 edg hi   Pos:IDT-00 Ref-0 edg hi
(0b)Cur:IDT-00 Ref-0 edg hi   Pos:IDT-00 Ref-0 edg hi
(0c)Cur:IDT-91 Ref-1 edg hi   Pos:IDT-00 Ref-0 edg hi
(0d)Cur:IDT-61 Ref-1 edg hi   Pos:IDT-00 Ref-0 edg hi
(0e)Cur:IDT-82 Ref-1 edg hi   Pos:IDT-00 Ref-0 edg hi
(0f)Cur:IDT-72 Ref-1 edg hi   Pos:IDT-00 Ref-0 edg hi
(10)Cur:IDT-51 Ref-3 lev low  Pos:IDT-00 Ref-0 edg hi
(11)Cur:IDT-b2 Ref-1 lev low  Pos:IDT-00 Ref-0 edg hi
(12)Cur:IDT-a2 Ref-5 lev low  Pos:IDT-00 Ref-0 edg hi
(13)Cur:IDT-92 Ref-1 lev low  Pos:IDT-00 Ref-0 edg hi
(14)Cur:IDT-62 Ref-2 lev low  Pos:IDT-00 Ref-0 edg hi
(15)Cur:IDT-a3 Ref-2 lev low  Pos:IDT-00 Ref-0 edg hi
 (16)Cur:IDT-b3 Ref-1 lev low  Pos:IDT-00 Ref-0 edg hi
 (17)Cur:IDT-52 Ref-1 lev low  Pos:IDT-00 Ref-0 edg hi
```

Как и ожидалось, IRQ 1 связан с IDT-записью 0x81.

Адрес ISR-процедуры для объекта прерывания хранится в поле ServiceRoutine (которое команда !idt показывает в своем выводе), а код прерывания, выполняемый при его возникновении, хранится в массиве DispatchCode в конце объекта прерывания. Хранящийся там код прерывания запрограммирован на создание фрейма системного прерывания в стеке и на последующий вызов функции, чей адрес хранится в поле DispatchAddress (KiInterruptDispatch в данном примере), с передачей этой функции указателя на объект прерывания. ■

Windows и обработка в режиме реального времени

Среда реального времени, аппаратная или программная, характеризуется требованиями выдерживания критического срока. Аппаратные системы реального времени (например, система управления атомной электростанцией) имеют критические сроки, которые должны соблюдаться системой, чтобы избежать катастрофических сбоев, приводящих к выходу из строя оборудования или к человеческим жертвам. Программные системы реального времени (например, автомобильные системы оптимизации расхода топлива) имеют критические сроки, которые могут быть пропущены системой, но своевременность по-прежнему является желаемым свойством. В системах реального времени у компьютеров есть сенсорные устройства ввода и управляющие устройства вывода. Конструктор компьютерных систем реального времени должен знать наихудшие значения задержек между временем генерирования прерывания входным устройством и временем возможного управления реакцией выходного устройства со стороны драйвера устройства. Этот анализ наихудшей ситуации должен принимать

в расчет задержки операционной системы, возникающие как из-за задержек приложения, так и из-за задержек, вносимых драйверами устройств.

Поскольку Windows не допускает управляемой установки приоритетов IRQ-запросов устройств и приложений пользовательского уровня только в случае пребывания процессорного IRQL-уровня в пассивном состоянии, Windows обычно не подходит в качестве операционной системы реального времени. В конечном счете, наибольшие задержки определяются не Windows, а устройствами системы и драйверами устройств. Этот фактор становится проблемой, когда разработчики систем реального времени используют какое-нибудь стандартное оборудование. У конструкторов могут возникнуть сложности в определении продолжительности наихудших задержек ISR или DPC стандартных устройств. Даже после тестирования конструктор не может гарантировать, что частный случай в живой системе не заставит эту систему пропустить критический срок. Более того, суммарное значение всех задержек системных DPC и ISR может существенно превышать значение, допустимое для чувствительной ко времени системы.

Хотя требования реального времени есть у многих типов встроенных систем (например, у принтеров и автомобильных компьютеров), у Windows Embedded Standard 7 характеристики реального времени отсутствуют. Это просто одна из разновидностей Windows 7, позволяющая выпускать компактные версии этой операционной системы, подходящие для запуска на устройствах с ограниченными ресурсами. Например, устройство без сетевых возможностей опустит все компоненты Windows 7, связанные с работой в сети, включая средства управления сетью, а также адаптер и драйвера устройств стека протокола.

Кроме того, существуют сторонние производители, поставляющие для Windows ядра реального времени. Подход, используемый ими, заключается во встраивании их ядра реального времени в специализированный HAL и в принуждении Windows запускаться в качестве задачи в операционной системе реального времени. Задача, запускающая Windows, служит в качестве пользовательского интерфейса к системе и имеет более низкий приоритет по сравнению с задачами, ответственными за управление устройством.

Связывание ISR с конкретным уровнем прерывания называется *подключением объекта прерывания*, а разобщение ISR и записи IDT называется *отключением объекта прерывания*. Эти операции, выполняемые путем вызова функций ядра IoConnectInterruptEx и IoDisconnectInterruptEx, позволяют драйверу устройства «включать» ISR при загрузке драйвера в систему и «выключать» ISR, если драйвер выгружается.

Использование объекта прерывания для регистрации ISR препятствует тому, чтобы драйверы устройства возились непосредственно с аппаратными средствами прерывания (имеющими отличия, связанные с архитектурами процессоров), и избавляет от необходимости знать обо всех подробностях IDT. Это свойство ядра помогает в создании переносимых драйверов устройств, поскольку оно исключает необходимость программировать на языке ассемблера или отражать разнообразие процессоров в драйверах устройств.

Объекты прерываний предоставляют также и другие преимущества. За счет использования объекта прерывания ядро может синхронизировать выполнение ISR с другими частями драйвера устройства, которые могут использовать с IRS общие данные.

Более того, объекты прерываний позволяют ядру легко вызывать более одной ISR-процедуры для любого уровня прерывания. Если объекты прерываний создаются несколькими драйверами устройств и подключаются к одной и той же записи IDT, диспетчер прерываний вызывает каждую процедуру при возникновении прерывания на указанной линии прерывания. Эта возможность позволяет ядру легко поддерживать конфигурации, составленные из последовательных цепей, в которых несколько устройств совместно используют одну и ту же линию прерывания. Цепь прерывается, когда одна из ISR-процедур заявляет о правах собственности на прерывание, возвращая статус диспетчеру прерываний.

Если одновременного обслуживания требуют несколько устройств, использующих одно и то же прерывание, то, как только диспетчер прерываний снизит IRQL, устройства, не получившие подтверждения от своих ISR-процедур, снова выставят прерывание системы. Выстраивание в цепочку будет разрешено, только если все драйверы устройств, ожидающие использования одного и того же прерывания, покажут ядру, что они могут совместно использовать это прерывание; если они этого сделать не могут, диспетчер устройств Plug and Play перестроит назначение прерываний, чтобы учесть требования по общему использованию прерываний каждого устройства. При наличии общего вектора прерывания объект прерывания вызывает функцию KiChainedDispatch, которая по очереди вызовет ISR-процедуры каждого зарегистрированного объекта прерывания, пока одна из них не заявит права на прерывание или пока все они не будут выполнены. В показанном выше примере вывода команды !idt («Эксперимент: просмотр IDT», с. 107) вектор 0xa2 подключен к нескольким выстроенным в цепочку объектам прерываний. Так сложилось, что на системе, где была запущена команда, этот вектор соответствовал встроенному карт-ридеру 7-в-1, представляющему собой комбинацию из устройств чтения флеш-карт Secure Digital (SD), Compact Flash (CF), MultiMedia Card (MMC) и карт других типов, и у каждого устройства имелось свое собственное прерывание. Поскольку они были сгруппированы производителем в одно устройство, вполне разумно было его прерываниям использовать один и тот же вектор.

СРАВНЕНИЕ ПРЕРЫВАНИЙ НА ОСНОВЕ ИСПОЛЬЗОВАНИЯ ЛИНИЙ И ПРЕРЫВАНИЙ, ИНИЦИИРУЕМЫХ СООБЩЕНИЯМИ

Общие прерывания часто являются причиной высокой латентности прерываний, а также могут стать причиной нестабильной работы системы. Обычно их использование нежелательно и выражается в побочном эффекте наличия на компьютере ограниченного количества физических линий прерывания. Например, в предыдущем примере с кард-ридером 7-в-1 намного лучше было бы, чтобы для каждого устройства было свое собственное прерывание и чтобы один драйвер управлял различными прерываниями, зная, от какого устройства пришло прерывание. Но расход четырех IRQ-линий на одно устройство быстро приводит к исчерпанию IRQ-линий. Кроме того, в любом случае каждое PCI-устройство подключается только к одной IRQ-линии, поэтому кард-ридер вообще не может использовать более одной IRQ-линии.

Еще одна проблема, связанная с генерированием прерываний по IRQ-линии, заключается в том, что неправильное управление IRQ-сигналом может привести к недопустимому пику прерываний (interrupt storms) или

к возникновению других разновидностей взаимных блокировок, поскольку пока ISR-процедура не подтвердит получение сигнала, он выставляется на «высоком» или «низком» уровне. Более того, контроллер прерываний должен, как правило, получать также и сигнал завершения прерывания EOI. Если какое-либо из этих событий из-за какого-нибудь сбоя не произойдет, система войдет в бесконечное состояние прерывания, или же следующие прерывания будут замаскированы, или же произойдет и то и другое. И наконец, прерывания на основе использования линий предоставляют плохую масштабируемость в мультипроцессорной среде. Во многих случаях оборудование принимает окончательное решение о том, работу какого процессора прервать из возможного набора, составленного из того, что отобрано для этого прерывания диспетчером устройств Plug and Play, и из того, что могут сделать небольшие драйверы устройств.

Решением всех этих проблем является новый механизм прерываний, который впервые был представлен в стандарте PCI 2.2 под именем *прерываний, инициируемых сообщениями* (Message-signaled interrupts, MSI). Хотя этот механизм остается необязательным компонентом стандарта, редко встречающимся на клиентских машинах, поддержка MSI, полностью поддерживаемая всеми последними версиями Windows, все чаще реализуется на серверах и рабочих станциях. В MSI-модели устройство доставляет сообщение своему драйверу, записывая его по конкретному адресу памяти. Это действие вызывает прерывание, а затем Windows вызывает ISR-процедуру с содержимым сообщения (значением) и адресом, по которому оно было доставлено. Устройство также может доставить по адресам памяти несколько сообщений (до 32), предоставляя на основе события различную полезную информацию.

Поскольку связь основана на значении памяти и поскольку содержимое доставляется вместе с прерыванием, надобность в IRQ-линиях отпадает (общий системный лимит MSI-прерываний делается равным количеству векторов прерываний, а не количеству IRQ-линий), отпадает также и необходимость в ведущей ISR-процедуре для запроса у устройства данных, связанных с прерыванием, что снижает задержку. В связи с доступностью в данной модели большого количества прерываний, связанных с устройствами, фактически сводится на нет вся польза от применения общих прерываний, а за счет непосредственной доставки данных прерывания заинтересованным в этом ISR-процедурам еще больше уменьшаются задержки.

И наконец, во вводимое в PCI 3.0 расширение MSI-модели MSI-X добавляется поддержка 32-разрядных сообщений (вместо 16-разрядных) с максимумом в 2048 различных сообщений (вместо всего лишь 32), и, что более важно, возможность использования другого адреса (который может определяться в динамическом режиме) для каждой полезной информации MSI. Использование другого адреса позволяет записывать полезную информацию MSI в другой физический диапазон адресов, который принадлежит другому процессору или другому набору целевых процессоров, что дает возможность эффективно использовать технологию доставки прерывания, осведомленного о доступе к неоднородной памяти (nonuniform memory access, NUMA) тому процессору, который инициировал связанный с этим прерыванием запрос к устройству. Это улучшает время ожидания и масштабируемость путем контролирования во время завершения прерывания и загруженности, и расположения ближайшего NUMA-узла.

РОДСТВЕННОСТЬ И ПРИОРИТЕТНОСТЬ ПРЕРЫВАНИЙ

На системах, поддерживающих ACPI и содержащих APIC, Windows позволяет разработчикам драйверов и администраторам отчасти управлять родственностью процессоров (выбирая процессор или группу процессоров, получающих прерывание) и политикой родственности (выбирая способ, с помощью которого произойдет отбор процессоров, и определится, какой процессор в группе будет выбран). Более того, он позволяет использовать примитивный механизм приоритетности прерываний на основе выбора IRQL. Политика родственности (affinity policy) определяется в соответствии с данными, показанными в табл. 3.1, и настраивается с помощью значения реестра под названием InterruptPolicyValue в подразделе Interrupt Management\Affinity Policy раздела реестра, представляющего устройство. Благодаря этому для настройки не требуется никакого кода — администратор, чтобы повлиять на поведение устройства, может добавить это значение к разделу этого устройства. Для всего этого Microsoft предоставляет средство под названием Interrupt Affinity policy Tool, которое может быть загружено с адреса http://www.microsoft.com/whdc/system/sysperf/intpolicy.mspx.

Таблица 3.1. Политика родственности IRQ

Политика	Значение
IrqPolicyMachineDefault	Устройству не требуется какая-то особая политика родственности. Windows использует политику, имеющуюся на машине по умолчанию, которая (на машинах с менее чем восемью логическими процессорами) заключается в выборе любого доступного на машине процессора
IrqPolicyAllCloseProcessors	На NUMA-машине диспетчер устройств Plug and Play назначает прерывание всем процессорам, близким к устройству (на том же узле). На машинах, не имеющих NUMA-доступа, эта политика аналогична политике IrqPolicyAllProcessorsInMachine
IrqPolicyOneCloseProcessor	На NUMA-машине диспетчер устройств Plug and Play назначает прерывание одному процессору, который ближе всех к устройству (на том же узле). На машинах, не имеющих NUMA-доступа, выбирается любой, доступный в системе процессор
IrqPolicyAllProcessorsInMachine	Прерывание обрабатывается любым доступным на машине процессором
IrqPolicySpecifiedProcessors	Прерывание обрабатывается только одним из процессоров, указанных в маске родственности в значении реестра AssignmentSetOverride
IrqPolicySpreadMessagesAcrossAllProcessors	Различные прерывания, инициируемые сообщениями, распределяются по оптимальному набору подходящих процессоров, сообразуясь, по возможности, с топологией NUMA. Для этого требуется поддержка MSI-X на устройстве и на платформе

Кроме установки этой политики родственности для установки приоритетности прерываний можно использовать еще одно значение реестра из тех, что показаны в табл. 3.2.

Как уже ранее говорилось, важно отметить, что Windows не является операционной системой реального времени, и поэтому эти приоритеты IRQ — просто советы, дающиеся системе, которая управляет только IRQL, связанным с прерыванием, и не предоставляет никакой дополнительной приоритетности, кроме имеющегося в Windows механизма схемы приоритетов. Поскольку приоритетность IRQ также хранится в реестре, администраторы имеют право устанавливать эти значения для драйверов с требованием более низкого уровня латентности для тех драйверов, которые не пользуются возможностью хранения данных в реестре.

Таблица 3.2. Приоритеты IRQ

Приоритет	Значение
IrqPriorityUndefined	Устройству никакая особая приоритетность не требуется. Оно принимает приоритеты, установленные по умолчанию (IrqPriorityNormal)
IrqPriorityLow	Для устройства вполне приемлемы большие задержки, и оно должно получить более низкий уровень IRQL, чем обычно
IrqPriorityNormal	Устройство ожидает среднее время задержки. Оно получает IRQL-уровень, устанавливаемый по умолчанию и связанный с его вектором прерывания
IrqPriorityHigh	Устройство требует как можно меньшего времени задержки. Оно получает повышенный уровень IRQL, превышающий обычно назначаемый уровень

Программные прерывания

Хотя большинство прерываний генерируется на аппаратном уровне, ядро Windows генерирует программные прерывания для решения множества задач, среди которых и нижеперечисленные:

- запуск диспетчеризации потоков;
- обработка не критичных по времени прерываний;
- обработка истечения времени таймера;
- асинхронное выполнение процедуры в контексте конкретного потока;
- поддержка асинхронных операций ввода-вывода.

Эти задачи рассматриваются в следующих разделах.

Диспетчеризируемые прерывания или прерывания отложенного вызова процедуры (Deferred Procedure Call, DPC). Когда выполнение потока больше не может продолжаться, возможно, из-за того, что он был завершен или из-за того, что он добровольно вошел в состояние ожидания, ядро напрямую вызывает диспетчер для немедленного переключения контекста. Но иногда ядро обнаруживает, что перепланирование должно произойти, когда выполняемый код имеет глубокое многоуровневое вложение. В такой ситуации ядро запрашивает диспетчеризацию, но откладывает ее наступление, покуда не завершит свою текущую работу. Использование программного DPC-прерывания является удобным способом реализации этой задержки.

Ядро всегда поднимает IRQL процессора до уровня DPC/dispatch или выше, когда ему нужно синхронизировать доступ к общим структурам ядра. Тем самым блокируются дополнительные программные прерывания и диспетчеризация потоков. Когда ядро обнаруживает необходимость диспетчеризации, оно запрашивает прерывание уровня DPC/dispatch, но процессор задерживает прерывание, поскольку IRQL находится на этом или на более высоком уровне. Когда ядро завершает свою текущую работу, процессор видит, что оно собирается поставить IRQL ниже уровня DPC/dispatch, и проверяет наличие каких-либо отложенных прерываний диспетчеризации. Если такие прерывания имеются, IRQL понижается до уровня DPC/dispatch, и происходит обработка прерываний диспетчеризации. Активизация диспетчера потоков с помощью программного прерывания является способом, позволяющим отложить диспетчеризацию, пока для нее не сложатся нужные обстоятельства. Но Windows использует программные прерывания для того, чтобы отложить и другие типы обработки.

Кроме диспетчеризации потоков ядро обрабатывает на этом уровне IRQL и отложенные вызовы процедур (DPC). DPC является функцией, выполняющей ту системную задачу, которая менее критична по времени, чем текущая задача. Функции называются *отложенными* (deferred), потому что они не требуют немедленного выполнения.

Отложенные вызовы процедур дают операционной системе возможность генерировать прерывание и выполнять системную функцию в режиме ядра. Ядро использует DPC-вызовы для обслуживания истечений времени таймера (и освобождения потоков, которые ожидают истечения времени таймеров) и для перепланирования времени использования процессора после истечения времени, выделенного потоку (кванта потока). Драйверы устройств используют DPC-вызовы для обработки прерываний. Для обеспечения своевременного обслуживания программных прерываний Windows совместно с драйверами устройств старается сохранять IRQL ниже IRQL-уровней устройств. Одним из способов достижения этой цели является выполнение ISR-процедурами драйверов устройств минимально необходимой работы для оповещения своих устройств, сохранения временного состояния прерывания и задержки передачи данных или обработки других, менее критичных по времени прерываний для выполнения в DPC-вызовах на IRQL-уровне DPC/dispatch.

DPC-вызов представлен *DPC-объектом*, который является объектом управления ядра, невидимым для программ пользовательского режима, но видимым для драйверов устройств и другого системного кода. Наиболее важной информацией, содержащейся в DPC-объекте, является адрес системной функции, которую ядро вызовет при обработке DPC-прерывания. DPC-процедуры, ожидающие выполнения, хранятся в очередях, управляемых ядром, — по одной очереди на каждый процессор. Они называются *DPC-очередями*. Для запроса DPC системный код вызывает ядро для инициализации DPC-объекта, а затем помещает этот объект в DPC-очередь.

По умолчанию ядро помещает DPC-объекты в конец DPC-очереди того процессора, при работе которого была запрошена DPC-процедура (обычно того процессора, на котором выполняется ISR-процедура). Но драйвер устройства может отменить такое поведение, указав DPC-приоритет (низкий, средний, выше среднего или высокий, где по умолчанию используется средний приоритет) и нацелив DPC на конкретный процессор. DPC-вызов, нацеленный на конкретный центральный процессор, известен как *целевой DPC*. Если DPC имеет высокий

приоритет, ядро ставит DPC-объект в начало очереди, в противном случае для всех остальных приоритетов оно ставит объект в конец очереди.

Ядро обрабатывает DPC-вызовы, когда IRQL-уровень процессора готов понизиться с IRQL-уровня DPC/dispatch или более высокого уровня до более низкого IRQL-уровня (APC или passive). Windows обеспечивает пребывание IRQL на уровне DPC/dispatch и извлекает DPC-объекты из очереди текущего процессора до тех пор, пока она не будет исчерпана (то есть ядро «расходует» очередь), вызывая по очереди каждую DPC-функцию. Ядро даст возможность IRQL-уровню упасть ниже уровня DPC/dispatch и позволить продолжить обычное выполнение потока только когда очередь истощится. Обработка DPC изображена на рис. 3.7.

DPC-приоритеты могут повлиять на поведение системы и другим образом. Обычно ядро инициирует расход DPC-очереди прерыванием DPC/dispatch-уровня. Ядро генерирует такое прерывание только в том случае, когда DPC-вызов направлен на текущий процессор (тот, на котором выполняется ISR-процедура) и DPC имеет приоритет выше низкого (low). Если DPC имеет низкий приоритет, ядро запрашивает прерывание только в том случае, когда количество невыполненных запросов DPC процессора превышает пороговое значение или если количество DPC-вызовов, выполнение которых запрашивается на процессоре в данном окне времени, невелико.

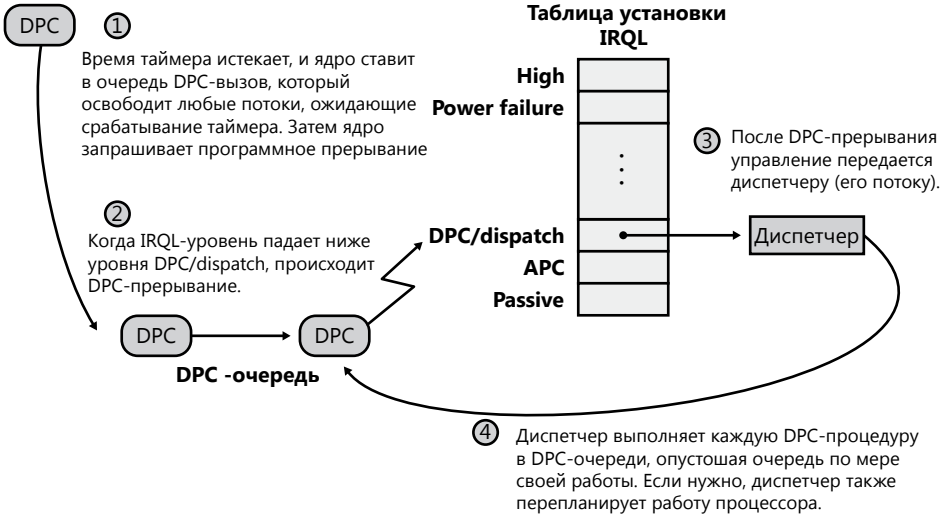


Рис. 3.7. Доставка DPC-вызова

Если DPC-вызов нацелен на центральный процессор, отличающийся от того, на котором запущена ISR-процедура, и приоритет DPC высокий (high) или выше среднего (medium-high), ядро немедленно сигнализирует целевому центральному процессору (посылая ему диспетчерское IPI) о необходимости расхода его DPC-очереди, но только если целевой процессор находится в режиме ожидания. Если приоритет средний или низкий, то для выдачи ядром прерывания DPC/dispatch количество DPC-вызовов, находящихся в очереди на целевом процессоре, должно превысить определенный порог. Системный поток простоя (idle) также опустошает DPC-очередь того процессора, на котором он запущен. Несмотря

на ту гибкость, которую придают системе целевые назначения DPC-вызовов и уровни приоритета, драйверам устройств редко требуется изменять поведение по умолчанию своих DPC-объектов. Ситуации, инициирующие опустошение DPC-очереди, сведены в табл. 3.3. Если посмотреть на правила генерации, то, фактически, получается, что приоритеты выше среднего и высокий равны друг другу. Разница проявляется при их вставке в список, когда прерывания высокого уровня находятся впереди, а прерывания уровня выше среднего сзади.

Поскольку потоки пользовательского режима выполняются при низком IRQL, высоки шансы на то, что DPC-вызов прервет выполнение обычного пользовательского потока. DPC-процедуры выполняются независимо от того, какой поток запущен, стало быть, когда запускается DPC-процедура, она не может выстроить предположение насчет того, чье адресное пространство, какого именно процесса в данный момент отображается. DPC-процедуры могут вызывать функции ядра, но они не могут вызывать системные службы, генерировать ошибки отсутствия страницы или же создавать или ожидать объекты диспетчеризации (рассматриваемые позже в этой главе). Тем не менее они могут обращаться к невыгружаемым адресам системной памяти, поскольку системное адресное пространство отображено всегда, независимо от того, что из себя представляет текущий процесс.

Таблица 3.3. Правила генерации DPC-прерывания

Приоритет DPC	DPC-вызов нацеливается на процессор, выполняющий ISR-процедуру	DPC-вызов нацеливается на другой процессор
Низкий (Low)	Длина DPC-очереди превышает максимальную длину DPC-очереди или уровень DPC-запросов ниже минимального уровня DPC-запросов	Длина DPC-очереди превышает максимальную длину DPC-очереди или система находится в простое
Средний (Medium)	Всегда	Длина DPC-очереди превышает максимальную длину DPC-очереди или система находится в простое
Выше среднего (Medium-High)	Всегда	Целевой процессор простаивает
Высокий (High)	Всегда	Целевой процессор простаивает

DPC-вызовы предоставляются в первую очередь драйверам устройств, но также используются и ядром. Чаще всего ядро использует DPC для обработки истечения кванта времени. При каждом такте системных часов возникает прерывание на IRQL-уровне clock. *Обработчик прерывания от часов* (запускаемый на IRQL-уровне clock) обновляет системное время и затем уменьшает показание счетчика, отслеживающего продолжительность работы текущего потока. Когда счетчик доходит до нуля, квант времени потока истекает, и ядру, возможно, нужно будет перепланировать время процессора, то есть выполнить задачу с более низким приоритетом, которая должна выполняться на IRQL-уровне DPC/dispatch. Обработчик прерывания от часов ставит DPC-вызов в очередь, чтобы инициировать диспетчеризацию потоков, а затем завершить свою работу и понизить IRQL-

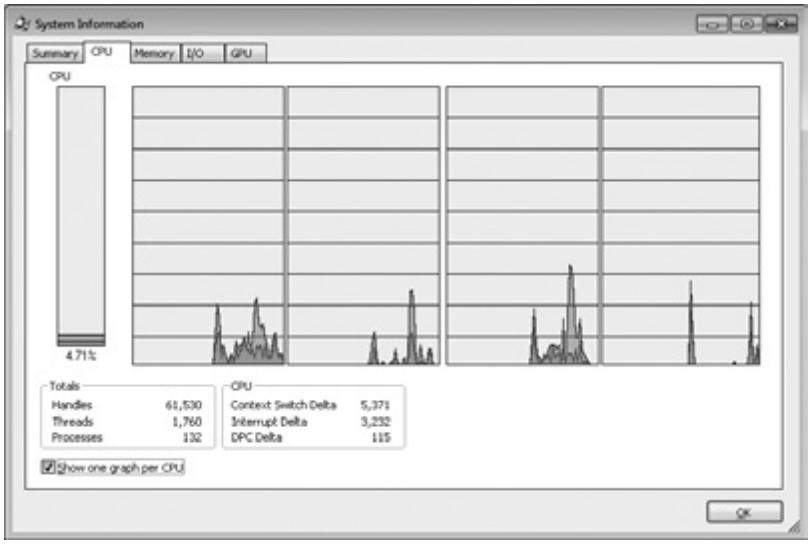
уровень процессора. Поскольку у DPC-прерывания приоритет ниже, чем у прерываний от устройств, любые отложенные прерывания от устройств, появляющиеся до завершения прерывания от часов, обрабатываются до выдачи DPC-прерывания.

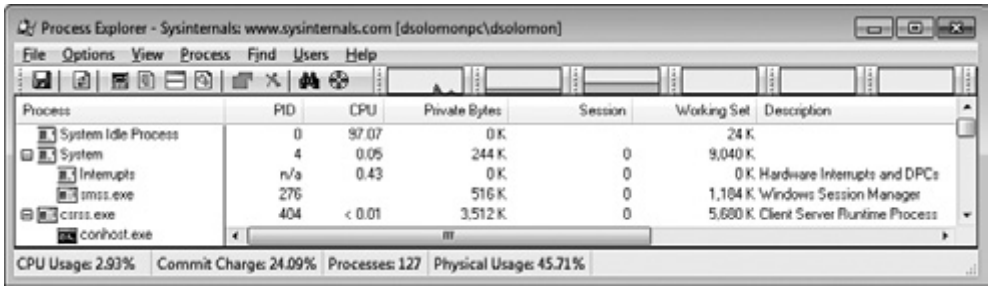
Так как DPC-вызовы выполняются независимо от того, какой из потоков выполняется в данное время на системе (что во многом похоже на прерывания), они являются основной причиной того, что система, на которой они выполняются, становится невосприимчивой к рабочей нагрузке клиентских систем или рабочей станции, потому что даже потоки с самым высоким уровнем приоритета будут прерваны ожидающим DPC-вызовом. Некоторые DPC-вызовы выполняются настолько долго, что пользователи могут заметить отставание видео или звука или даже ощутить ненормальное замедление реакции мыши или клавиатуры, поэтому для драйверов с продолжительными DPC-вызовами Windows поддерживает *потоковые DPC-вызовы*.

Потоковые DPC-вызовы, как следует из их названия, предназначены для выполнения DPC-процедуры на пассивном (passive) уровне в потоке с приоритетом реального времени (priority 31). Это позволяет DPC-вызову воспользоваться приоритетом над большинством потоков пользовательского режима (поскольку большинство потоков приложений не запускается в диапазонах приоритетов реального времени). Но это позволяет другим прерываниям, непотоковым DPC-вызовам, APC-вызовам и потокам с более высоким приоритетом реализовать свой приоритет перед данной процедурой.

ЭКСПЕРИМЕНТ: ОТСЛЕЖИВАНИЕ АКТИВНОСТИ ПРЕРЫВАНИЙ И DPC

Для отслеживания активности прерываний и DPC можно воспользоваться средством Process Explorer, открыв диалоговое окно System Information (Системная информация) и перейдя во вкладку CPU (Центральный процессор), где будет показано количество прерываний и DPC-процедур, зафиксированное при каждом обновлении средством Process Explorer отображаемых результатов (по умолчанию с периодичностью в одну секунду).





Можно также отследить выполнение конкретных процедур обслуживания прерываний и отложенных вызовов процедур, используя встроенное отслеживание событий (рассматриваемое далее в этой главе):

1. Запустите перехват событий. Откройте окно командной строки, перейдите в каталог Microsoft Windows Performance Toolkit (обычно он находится в каталоге c:\Program Files) и наберите следующую команду[1]:

```
xperf -on PROC_THREAD+LOADER+DPC+INTERRUPT
```

2. Остановите перехват событий, набрав следующую команду:

```
xperf -d dpcisr.etl
```

3. Сгенерируйте отчеты для перехвата событий, набрав следующее:

```
xperf dpcisr.etl
tracerpt \kernel.etl -report dpcisr.html -f html
```

   В результате будет сгенерирована веб-страница dpcisr.html.

4. Откройте файл report.html и раскройте подраздел DPC/ISR. Раскройте область DPC/ISR Breakdown и увидите сводку, показывающую время, затраченное на ISR-процедуры и DPC-вызовы каждым драйвером. Например, как на рисунке на странице напротив.

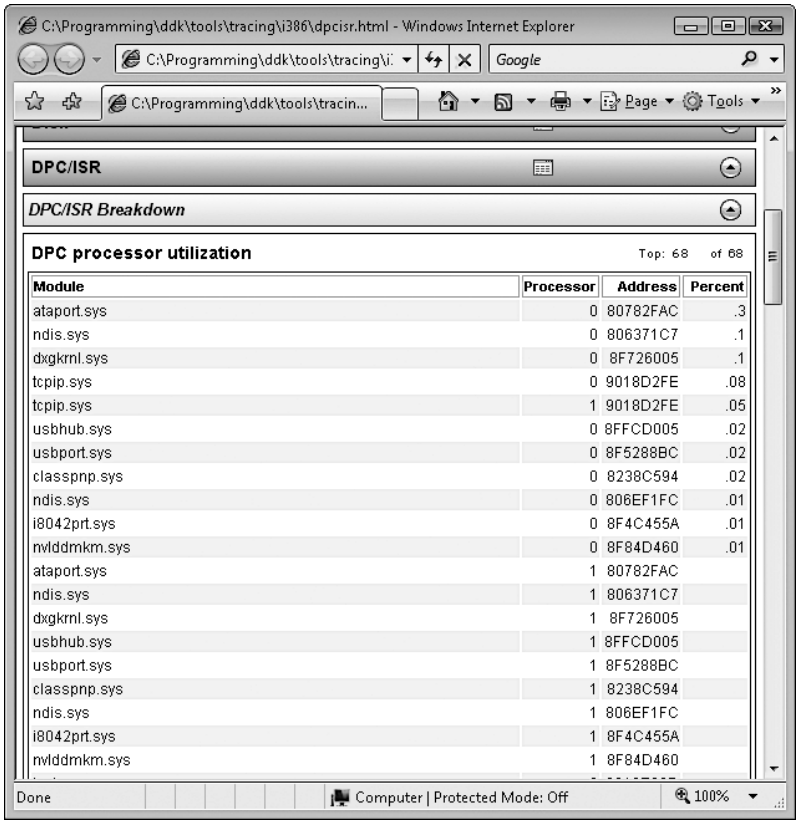
При запуске в отладчике ядра команды ln с указанием адреса каждой записи о событии показывается имя функции, выполняющей DPC или ISR:

```
lkd> ln 0x806321C7
(806321c7)   ndis!ndisInterruptDpc
lkd> ln 0x820AED3F
(820aed3f)   nt!IopTimerDispatch
lkd> ln 0x82051312
(82051312)   nt!PpmPerfIdleDpc
```

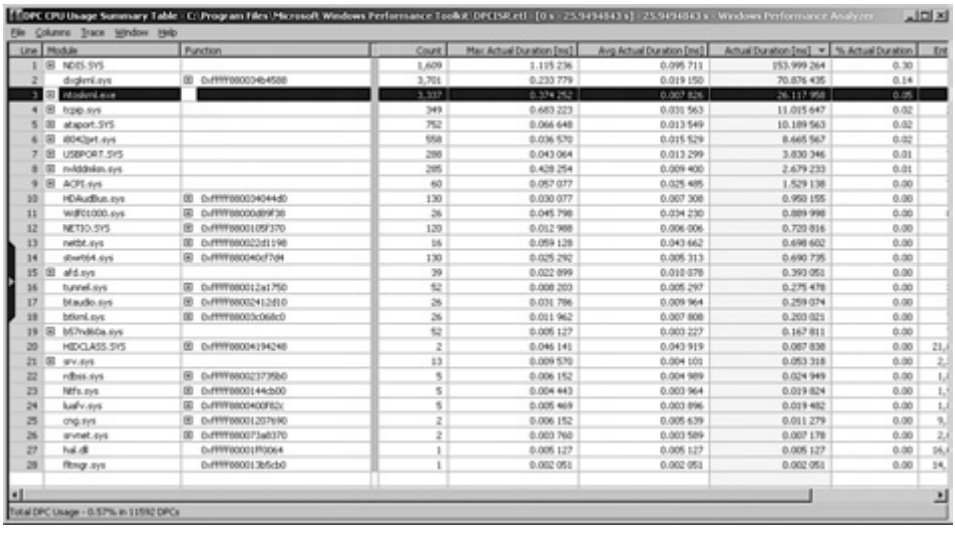
Первым показан DPC-вызов, помещенный в очередь драйвером мини-порта сетевой карты NDIS. Вторым показан DPC-вызов обработки истечения времени общего таймера ввода-вывода. Третий адрес относится к DPC-вызову, предназначенному для выполнения операции простоя (idle).

Кроме использования этого средства для получения отчета в формате HTML, для просмотра подробного обзора всех DPC- ISR-событий можно воспользоваться средством Xperf Viewer, щелкнув в главном окне Xperf правой кнопкой мыши на пункте DPC and/or ISR CPU Usage graphs (Диа-

[1] Перед этим нужно убедиться в том, что никакая другая программа, такая как Process Explorer или Process Monitor, не перехватывает события, иначе выполнение команды завершится с ошибкой.



граммы использования центрального процессора DPC и ISR) и выбрав пункт Summary Table (Сводная таблица). Вам будет дана возможность детально рассмотреть каждые DPC и ISR каждого драйвера, а также увидеть продолжительность и количество, как показано на следующем рисунке.



Механизм потоковых DPC-вызовов включен по умолчанию, но его можно отключить путем добавления нулевого DWORD-значения в параметр HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\kernel\ThreadDpcEnable. Поскольку потоковые DPC-вызовы могут быть выключены, разработчики драйверов, пользующиеся потоковыми DPC-вызовами, должны писать свои процедуры, следуя тем же правилам, которые распространяются на непотоковые вызовы. Эти процедуры не могут обращаться к выгружаемой памяти, ждать диспетчеризации или выстраивать предположения насчет IRQL-уровня, с которым они выполняются. Кроме того, они не должны использовать API-функции KeAcquire/ReleaseSpinLockAtDpcLevel, поскольку функции предполагают, что центральный процессор имеет уровень dispatch. Вместо этого в потоковых DPC-вызовах нужно использовать функцию KeAcquire/ReleaseSpinLockForDpc, выполняющую соответствующее действие после проверки текущего IRQL.

Прерывания асинхронных вызовов процедур. Асинхронные вызовы процедур (Asynchronous procedure call, APC) дают пользовательским программам и системному коду способ выполнения в контексте конкретного пользовательского потока (а следовательно, в адресном пространстве конкретного процесса). Поскольку APC-вызовы выстраиваются в очередь на выполнение в контексте конкретного потока и запускаются на IRQL-уровне, который ниже уровня DPC/dispatch, они не подпадают под такие же ограничения, которые накладываются на DPC. APC-процедура может получать ресурсы (объекты), ждать дескрипторов объектов, справляться с ошибками отсутствия страницы и вызывать системные службы.

APC-вызовы описываются объектом управления ядра, называемым *APC-объектом*. APC-вызовы, ожидающие выполнения, помещаются в управляемую ядром *APC-очередь*. В отличие от очереди DPC, которая видна всей системе, APC-очередь относится к конкретному потоку — у каждого потока есть своя собственная APC-очередь. При запросе на помещение в очередь APC-вызова ядро вставляет ее в очередь, принадлежащую тому потоку, который будет выполнять APC-процедуру. Ядро, в свою очередь, запрашивает программное прерывание на APC-уровне а затем, когда поток через некоторое время начнет работать, в нем выполняется APC-вызов.

Существует два вида APC-вызовов: режима ядра и пользовательского режима. APC-вызовы режима ядра не требуют разрешения от целевого потока на запуск в его контексте, а APC-вызовы пользовательского потока требуют такого разрешения. APC-вызовы режима ядра прерывают поток и выполняются без его вмешательства или разрешения. Есть также два вида APC-вызовов режима ядра: обычные и специальные. Специальные APC-вызовы выполняются на уровне APC и позволяют APC-процедуре изменять некоторые APC-параметры. Обычные APC-вызовы выполняются на уровне passive и получают измененные параметры от специальной APC-процедуры (или исходные параметры, если они не были изменены).

Обычные и специальные APC-вызовы могут быть отключены путем повышения IRQL на APC-уровень или путем вызова процедуры KeEnterGuardedRegion. Она отключает APC-доставку, устанавливая поле SpecialApcDisable в структуре KTHREAD вызывающего потока (этот механизм рассматривается в главе 5). Поток может отключить обычные APC-вызовы только путем вызова процедуры KeEnterCriticalRegion, которая устанавливает поле KernelApcDisable в структуре

KTHREAD потока. В табл. 3.4 сведено поведение каждого вида APC-вызова по вставке и доставке APC.

Таблица 3.4. Вставка и доставка APC

Вид APC-вызова	Поведение, связанное со вставкой	Поведение, связанное с доставкой
Специальный (режим ядра)	Вставляется в конец списка APC-вызовов режима ядра	Доставляется на APC-уровне, как только понизится IRQL, и при условии, что поток не находится в защищенной области. Даются указатели на аргументы, определенные при вставке APC-вызова
Обычный (режим ядра)	Вставляется сразу же после последнего специального APC-вызова (во главе всех остальных обычных APC-вызовов)	Доставляется на уровне PASSIVE_LEVEL после выполнения связанного специального APC-вызова. Доставке задаются аргументы, возвращенные связанным специальным APC-вызовом (это могут быть исходные аргументы, использованные при вставке, или новые аргументы)
Обычный (пользовательский режим)	Вставляется в конец списка APC-вызовов пользовательского режима	Доставляется на уровне PASSIVE_LEVEL, как только понизится IRQL, и при условии, что поток не находится в критической (или защищенной) области, а также если поток находится в состоянии готовности. Доставке задаются аргументы, возвращенные связанным специальным APC-вызовом (это могут быть исходные аргументы, использованные при вставке, или новые аргументы)
Обычный (пользовательский режим) Выход из потока (PsExitSpecialApc)	Вставляется в начало списка APC-вызовов пользовательского режима	Доставляется на уровне PASSIVE_LEVEL по возвращении в пользовательский режим, если находится в готовности к выполнению в состоянии ожидания. Доставке задаются аргументы, возвращенные специальным APC-вызовом, завершающим поток

Исполняющая система использует APC-вызовы режима ядра для выполнения работы операционной системы, которая должна быть завершена в адресном пространстве (в контексте) конкретного потока. Она может использовать специальные APC-вызовы, чтобы направить поток, к примеру, на остановку выполнения прерываемой системной службы или для записи результатов асинхронной операции ввода-вывода в адресном пространстве потока. Подсистемы среды окружения используют специальные APC-вызовы режима ядра, чтобы заставить поток приостановить или завершить свою работу, или же получить или установить контекст его выполнения в пользовательском режиме. Подсистема для UNIX-приложений использует APC-вызовы режима ядра для имитации доставки UNIX-сигналов процессам подсистемы для UNIX-приложений.

Другое важное применение APC-вызовов режима ядра относится к приостановке или завершению потока. Поскольку эти операции могут инициироваться произвольными потоками и направлены на другие произвольные потоки, ядро

использует APC для запроса контекста потока, а также для завершения потока. Драйверы устройств часто блокируют APC-вызовы или входят в критическую или охраняемую область, чтобы воспрепятствовать выполнению этих операций в тот момент, когда они удерживают блокировку, в противном случае блокировка может быть никогда не снята, и система зависнет.

APC-вызовы режима ядра используются также драйверами устройств. Например, если инициирована операция ввода-вывода и поток перешел в режим ожидания, может быть спланирован запуск другого потока в другом процессе. Когда устройство завершит передачу данных, система ввода-вывода должна каким-то образом вернуться в контекст потока, инициировавшего ввод-вывод, чтобы он мог скопировать результаты операции ввода-вывода в буфер в адресном пространстве процесса, содержащего этот поток. Для выполнения этого действия система ввода-вывода использует специальный APC-вызов режима ядра, если только приложение не использовало API-функция SetFileIoOverlappedRange или порты завершения ввода-вывода, — в таком случае либо буфер в памяти будет глобальным, либо копирование его произойдет только после того, как поток извлечет из порта признак завершения.

APC-вызовы пользовательского режима используются несколькими Windows API-функциями, такими как ReadFileEx, WriteFileEx и QueueUserAPC. Например, функции ReadFileEx и WriteFileEx позволяют вызывающему коду указать подпрограмму завершения, вызываемую при окончании операции ввода-вывода. Завершение ввода-вывода реализуется постановкой APC-вызова в очередь того потока, который выдал запрос на ввод-вывод. Но обратный вызов процедуры завершения не обязательно происходит при постановке APC-вызова в очередь, поскольку APC-вызовы пользовательского режима доставляются потоку только в том случае, когда он находится *в готовности к работе в режиме ожидания*. Поток может войти в режим ожидания либо в ожидании дескриптора объекта и обозначении, что его ожидание ведется в готовности к работе (с помощью Windows-функции WaitForMultipleObjectsEx), либо путем непосредственной проверки на наличие отложенного APC (с помощью функции SleepEx). В обоих случаях, если APC-вызов пользовательского режима отложен, ядро прерывает (извещает) поток, передавая управление APC-процедуре, и возобновляет выполнение потока, когда APC-процедура завершит свою работу. В отличие от APC-вызовов режима ядра, которые могут выполняться на уровне APC, APC-вызовы пользовательского режима выполняются на уровне passive.

Доставка APC может изменить порядок очередей ожидания, то есть изменить списки, в которых указано, какие потоки, что и в каком порядке ожидают. Если при доставке APC-вызова поток находится в состоянии ожидания, после завершения APC-процедуры ожидание повторно выставляется или выполняется. Если ожидание все еще не разрешено, поток возвращается в состояние ожидания, но теперь он будет в конце списка в отношении тех объектов, которые им ожидаются. Например, поскольку APC-вызовы используются для приостановки выполнения потока, если поток ожидает каких-нибудь объектов, его ожидание удаляется до тех пор, пока не возобновится выполнение потока, после чего этот поток будет в конце списка потоков, ожидающих доступа к объектам, которых он ждет. Поток, выполняющий ожидание в готовности к работе в режиме ядра, будет также разбужен при завершении своей работы. Это позволит такому

потоку проверить, разбужен ли он в результате завершения своей работы или по какой-то другой причине.

Обработка таймера

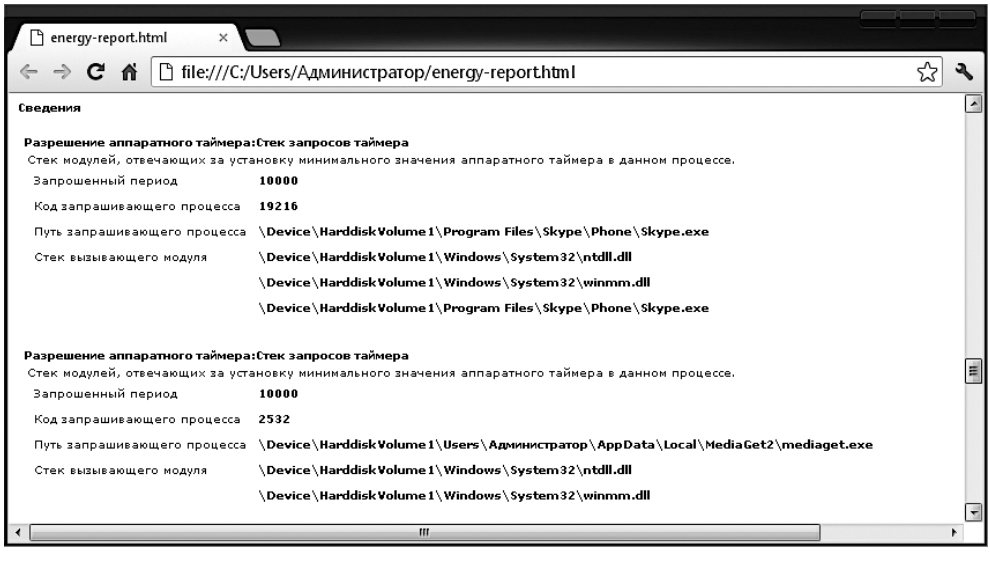
Интервальный таймер системных часов — пожалуй, наиболее важное устройство на Windows-машине, о чем свидетельствует его высокое значение IRQL (CLOCK_LEVEL). Его важность также следует из важности той работы, за которую он отвечает. Без прерывания от таймера Windows не сможет отслеживать время, что выразится в ошибочных результатах доступного времени и показаний часов и, что еще хуже, приведет к тому, что время таймеров больше не будет истекать, и потоки никогда больше не израсходуют свои кванты времени. Windows также перестанет быть вытесняющей операционной системой, и, пока текущий работающий поток не уступит центральный процессор, важные фоновые задачи и планировщики никогда не будут работать на этом процессоре.

Windows программирует системные часы на выдачу сигнала через наиболее подходящий для машины интервал и впоследствии позволяет драйверам, приложениями и администраторам изменять этот интервал в соответствии с их потребностями. Обычно системные часы обслуживаются либо через программируемый интервальный таймер (Programmable Interrupt Timer, PIT), микросхему, которая имеется на всех компьютерах, начиная с PC/AT, либо через часы реального времени (Real Time Clock, RTC). PIT работает на микросхеме, которая настроена на одну треть несущей частоты цвета стандарта NTSC (изначально она использовалась для телевизионного выхода на первых видеокартах CGA), а матричная логика с пользовательским программированием (HAL) использует различные доступные умножители для получения интервалов с шагом в 1 мс на модуль, начиная с 1 мс и заканчивая 15 мс. RTC запускаются на частоте 32,768 КГц, которая, будучи степенью числа 2, легко настраивается для работы с различными интервалами, также являющимися степенями числа 2. На современных машинах APIC Multiprocessor HAL настраивает RTC на выдачу сигнала каждые 15,6 миллисекунды, что составляет примерно 64 раза в секунду. Некоторым видам Windows-приложений, таким как мультимедийные программы, требуется очень короткое время отклика. Фактически, некоторые мультимедийные задачи требуют показателей на уровне не выше 1 мс. Поэтому в Windows реализованы API-функции и механизмы, позволяющие снизить интервал прерывания от системных часов, что выражается в большем количестве прерываний (по крайней мере, на процессоре с нулевым номером). Следует заметить, это повышает разрешение всех таймеров в системе, что может привести к более частому истечению времени на других таймерах.

Windows при любой возможности старается вернуть таймер часов к его исходному значению. При каждом запросе изменения интервала часов со стороны процесса Windows увеличивает значение внутреннего счетчика ссылок и связывает его с процессом. Аналогично этому драйверы (которые также могут изменить частоты следования импульсов) добавляются в глобальный счетчик ссылок. Когда все драйверы восстановят работу часов и все процессы, изменившие их работу, либо завершатся, либо все восстановят, Windows возвращает настройку часов к ее исходному значению (или запрещает такое возвращение при переходе к более высокому значению, требуемому процессом или драйвером).

ЭКСПЕРИМЕНТ: ИДЕНТИФИКАЦИЯ ТАЙМЕРОВ, РАБОТАЮЩИХ С ПОВЫШЕННОЙ ЧАСТОТОЙ

В связи с проблемами, которые могут возникать из-за таймеров, работающих с повышенной частотой, Windows использует средство Event Tracing for Windows (ETW) для отслеживания всех процессов и драйверов, требующих изменения интервала системных часов с выводом времени события и требуемого интервала. Это средство показывает также и текущий интервал. Эти данные пригодятся как разработчикам, так и системным администраторам при выяснении причин низкой производительности батареи на системах, которые в иных условиях работали вполне исправно, а также для уменьшения суммарной потребляемой мощности на больших системах. Для получения этих данных нужно просто запустить команду powercfg /energy и будет выдан HTML-файл energy-report.html похожий на этот.



Прокрутите страницу вниз до раздела Разрешение аппаратного таймера (Platform Timer Resolution) и получите сведения обо всех приложениях, изменивших разрешение таймера и находящихся в активном состоянии, вместе с записями стека вызовов, показывающими причины данного вызова. Разрешения таймера показаны в сотнях наносекунд, следовательно, период в 20 000 соответствует 2 мс. В показанном примере оба приложения запрашивают более высокое разрешение.

Для получения данной информации можно также воспользоваться отладчиком. Для каждого процесса структура EPROCESS содержит ряд полей, показанных далее, которые помогают определить изменения в разрешении таймера:

```
+0x4a8 TimerResolutionLink : _LIST_ENTRY [ 0xfffffa80'05218fd8 - 0xfffffa80'059cd508 ]
+0x4b8 RequestedTimerResolution : 0
+0x4bc ActiveThreadsHighWatermark : 0x1d
+0x4c0 SmallestTimerResolution : 0x2710
+0x4c8 TimerResolutionStackRecord : 0xfffff8a0'0476ecd0 _PO_DIAG_STACK_RECORD
```

Следует заметить, что отладчик показывает дополнительный блок информации: наименьшее разрешение таймера, которое когда-либо было запрошено данным процессом. В этом примере показан процесс, относящийся к PowerPoint 2010, который обычно запрашивает более низкое разрешение таймера при показе слайдов, но это не относится к работе в режиме редактирования слайдов. EPROCESS-поля PowerPoint, показанные в предыдущем коде, служат доказательством этому, и стек можно проанализировать путем вывода дампа структуры PO_DIAG_STACK_RECORD.

И наконец, поле TimerResolutionLink соединяет все процессы внесшие изменения в разрешение таймера через список с двойными связями ExpTimerResolutionListHead. Анализ этого списка с помощью команды отладчика !list может выявить все процессы в системе, которые внесли или вносили изменения в разрешение таймера, когда команда powercfg недоступна или требуется информация о прошедших процессах:

```
lkd> !list "-e -x \"dt nt!_EPROCESS @$extret-@@(#FIELD_OFFSET(nt!_EPROCESS,
TimerResolutionLink))
ImageFileName SmallestTimerResolution RequestedTimerResolution\"
nt!ExpTimerResolutionListHead"

dt nt!_EPROCESS @$extret-@@(#FIELD_OFFSET(nt!_EPROCESS, TimerResolutionLink))
ImageFileName
SmallestTimerResolution RequestedTimerResolution
   +0x2e0 ImageFileName            : [15]  "audiodg.exe"
   +0x4b8 RequestedTimerResolution : 0
   +0x4c0 SmallestTimerResolution  : 0x2710

dt nt!_EPROCESS @$extret-@@(#FIELD_OFFSET(nt!_EPROCESS, TimerResolutionLink))
ImageFileName
SmallestTimerResolution RequestedTimerResolution
   +0x2e0 ImageFileName            : [15]  "chrome.exe"
   +0x4b8 RequestedTimerResolution : 0
   +0x4c0 SmallestTimerResolution  : 0x2710

dt nt!_EPROCESS @$extret-@@(#FIELD_OFFSET(nt!_EPROCESS, TimerResolutionLink))
ImageFileName
SmallestTimerResolution RequestedTimerResolution
   +0x2e0 ImageFileName            : [15]  "calc.exe"
   +0x4b8 RequestedTimerResolution : 0
   +0x4c0 SmallestTimerResolution  : 0x2710

dt nt!_EPROCESS @$extret-@@(#FIELD_OFFSET(nt!_EPROCESS, TimerResolutionLink))
ImageFileName
SmallestTimerResolution RequestedTimerResolution
   +0x2e0 ImageFileName            : [15]  "devenv.exe"
   +0x4b8 RequestedTimerResolution : 0
   +0x4c0 SmallestTimerResolution  : 0x2710
```

продолжение ⇩

```
dt nt!_EPROCESS @$extret-@@(#FIELD_OFFSET(nt!_EPROCESS, TimerResolutionLink))
ImageFileName
SmallestTimerResolution RequestedTimerResolution
   +0x2e0 ImageFileName            : [15]  "POWERPNT.EXE"
   +0x4b8 RequestedTimerResolution : 0
   +0x4c0 SmallestTimerResolution  : 0x2710

dt nt!_EPROCESS @$extret-@@(#FIELD_OFFSET(nt!_EPROCESS, TimerResolutionLink))
ImageFileName
SmallestTimerResolution RequestedTimerResolution
   +0x2e0 ImageFileName            : [15]  "winvnc.exe"
   +0x4b8 RequestedTimerResolution : 0x2710
   +0x4c0 SmallestTimerResolution  : 0x2710
```

Истечение времени таймера

Как уже говорилось, основной задачей ISR-процедуры, связанной с прерыванием, генерируемым RTC или PIT, является отслеживание системного времени, которое в основном выполняется процедурой KeUpdateSystemTime. Ее второе задание заключается в отслеживании логического времени выполнения, например времени выполнения процесса или потока, и системного *времени такта*, которое является базовым показателем, используемым такими API-функциями, как GetTickCount, которые, в свою очередь, используются разработчиками для операций со временем в их приложениях. Эта часть работы выполняется процедурой KeUpdateRunTime. Но перед выполнением любой из этих работ, процедура KeUpdateRunTime проверяет, не истекло ли время какого-либо таймера.

Таймеры Windows могут быть либо *абсолютными таймерами*, что предполагает в будущем определенный показатель истечения времени, либо *относительными таймерами*, которые содержат отрицательное значение истечения срока, используемое как положительное смещение от текущего времени при вводе таймера в действие. Внутри системы все таймеры преобразуются к абсолютному времени истечения срока, хотя система следит за тем, является ли это время «настоящим» абсолютным временем или преобразованным относительным временем. Эта разница важна в некоторых сценариях, например в таких, как переход на летнее время (или даже изменение показаний часов вручную). Если пользователь перевел часы с 13 часов на 19 часов, абсолютный таймер все равно выдаст сигнал в «20 часов», а вот относительный таймер, настроенный, скажем, на истечение времени «через 2 часа», не почувствует влияния изменения показания часов, поскольку 2 часа еще не прошли. При возникновении подобных событий изменения системного времени ядро перепрограммирует абсолютное время, связанное с относительными таймерами, чтобы оно соответствовало новым установкам.

Поскольку часы выдают сигнал через известный кратный интервал, нижние разряды текущего системного времени будут в одной из 64 известных позиций (на APIC HAL). Windows использует это обстоятельство для сведения всех таймеров драйверов и приложений в связанные списки на основе массива, в котором каждый элемент соответствует возможному кратному показателю

системного времени. Эта таблица, называемая *таблицей таймера*, находится в блоке управления областью процессора — PRCB, что позволяет каждому процессору представлять свое собственное независимое истечение времени таймера без необходимости запроса глобальной блокировки (рис. 3.8). Далее будет показано, чем определяется таблица таймера логического процессора, в которую вставляется таймер. Поскольку таблица таймера есть у каждого процессора, работа по истечении времени таймера проделывается каждым процессором. При инициализации каждого процессора таблица заполняется абсолютными таймерами с бесконечным сроком истечения времени, дабы избежать некогерентного состояния. Каждый кратный показатель системного времени, с которым может быть связан таймер, называется *исполнителем* и хранится в заголовке диспетчера объекта таймера. Поэтому, чтобы определить, истекло ли время таймера, требуется только лишь проверить, имеются ли в связанном списке какие-либо таймеры, связанные с текущим исполнителем.

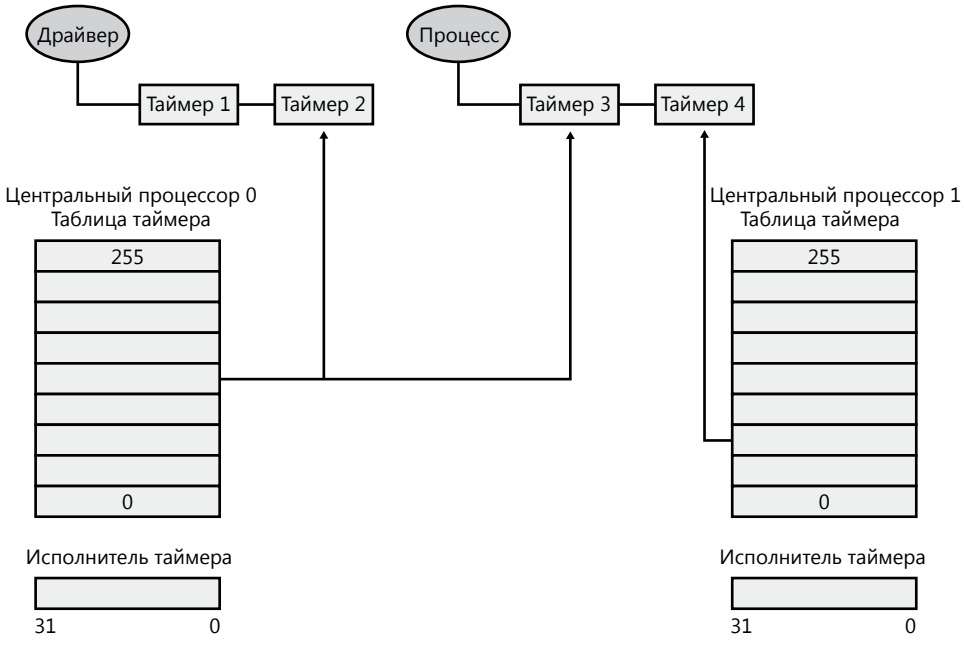


Рис. 3.8. Пример таймерных списков для каждого процессора

Хотя обновление счетчиков и проверка связанного списка являются быстрыми операциями, проход по каждому таймеру и изменение срока истечения его времени представляет из себя потенциально затратную операцию, ведь следует иметь в виду, что вся эта работа в настоящее время выполняется на уровне CLOCK_LEVEL, то есть на исключительно высоком IRQL-уровне. Аналогично тому, как ISR-процедура драйвера выстраивает в очередь DPC, чтобы отсрочить работу, ISR-процедура часов запрашивает программное прерывание DPC, устанавливает флаг в PRCB, чтобы механизм истощения DPC знал о том, что тай-

меры нуждаются в истечении срока своей работы. Точно так же, при обновлении времени выполнения процесса или потока, если ISR-процедура часов определяет, что у потока закончился квант времени, она также ставит в очередь программное прерывание DPC и устанавливает другой PRCB флаг. Эти флаги имеются у каждой PRCB-области, поскольку каждый процессор обычно совершает свои собственные вычисления по обновлению времени выполнения, так как каждый процессор выполняет индивидуальный поток и имеет различные связанные с ним задачи. В табл. 3.5 показаны различные поля, используемые при обработке таймера и истечении срока его работы.

Как только IRQL в рамках обработки DPC снова снизится до уровня DISPATCH_LEVEL, эти два флага будут установлены.

Таблица 3.5. Поля KPRCB, относящиеся к обработке таймера

KPRCB	Тип поля	Описание
ReadySummary	Поразрядная маска (32 разряда)	Поразрядная маска уровней приоритетов, имеющих один или несколько готовых потоков
DeferredReadyListHead	Список с одиночной связью	Начало списка с одиночной связью для отложенной очереди готовности
DispatcherReadyListHead	Массив с 32 списочными записями	Начала списков для 32 очередей готовности

В главе 5 рассматриваются действия, связанные с планированием потоков и истечением квантов времени. А здесь будет рассмотрена работа по истечении времени таймера. Поскольку таймеры связаны друг с другом исполнителем, код истечения времени (исполняемый DPC-вызовом, связанным с PRCB в поле TimerExpiryDpc) проводит разбор этого списка с головы до хвоста. (Во время ставки таймеры, ближайшие к значению, кратному интервалу часов, будут первыми, а за ними последуют таймеры, которые все ближе и ближе к следующему интервалу, но еще находящиеся в пределах данного исполнителя.) Для истечения времени таймера нужно выполнить две основные задачи:

- Таймер рассматривается как объект синхронизации диспетчера (потоки ждут таймера в рамках истечения лимита времени или непосредственно в рамках ожидания). На таймере будут запущены алгоритмы тестирования ожидания и удовлетворения ожидания. Эта работа рассмотрена далее в разделе, посвященном синхронизации. Именно таким образом таймер используется приложениями пользовательского режима и некоторыми драйверами.
- Таймер рассматривается как объект управления, связанный с процедурой обратного DPC-вызова, которая выполняется по истечении времени таймера. Этот метод зарезервирован только для драйверов и обеспечивает очень быструю реакцию на истечение времени таймера. (Метод ожидания/диспетчеризации требует задействовать всю дополнительную логику сигнализации ожидания.) Кроме того, поскольку само истечение времени таймера выполняется на уровне DISPATCH_LEVEL, на котором также запускаются DPC-процедуры, для процедуры обратного вызова таймера складываются весьма благоприятные обстоятельства.

Как только каждый процессор привлекается к обслуживанию таймера тактового интервала для выполнения обработки системного времени и времени выполнения, он также обрабатывает истечение времени таймера после небольшой задержки, во время которой IRQL-уровень снижается с CLOCK_LEVEL до DISPATCH_LEVEL. На рис. 3.9 эта работа показана на примере двух процессоров — сплошные стрелки показывают выдачу сигнала прерывания от часов, а пунктирные стрелки показывают обработку истечения времени таймера, которая может проводиться, если у процессора есть связанные с ним таймеры.

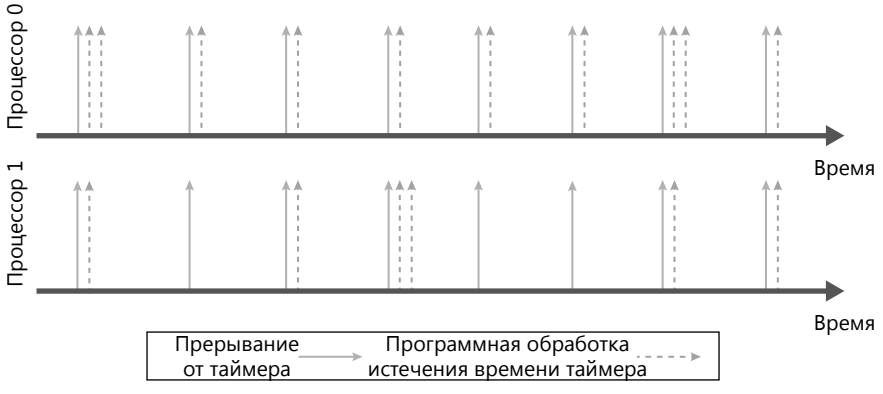


Рис. 3.9. Истечение времени таймера

Выбор процессора

При вставке таймера нужно принять одно важное решение, касающееся выбора подходящей таблицы, — иными словами, нужно выбрать наиболее оптимальный процессор. Если у таймера нет связанного с ним DPC-вызова, ядро сканирует все незапаркованные процессоры в текущей группе процессоров (о парковке можно прочитать в главе 5). Если текущий процессор запаркован, выбирается следующий процессор группы, в противном случае используется текущий процессор. С другой стороны, если с таймером связан какой-нибудь DPC-вызов, код вставки просто останавливает выбор на целевом процессоре, который связан с DPC, и выбирает таблицу таймера этого процессора.

В том случае, если разработчик драйвера не указал целевой процессор для DPC, ядро должно сделать свой выбор. Поскольку разработчики драйверов обычно ожидают, что DPC-вызов будет выполнен на том же самом процессоре, на котором в момент вставки был запущен код драйвера, ядро обычно выбирает центральный процессор 0. Именно он является процессором, следящим за временем, который всегда будет активен для обработки прерываний от часов. Но на серверных системах ядро выбирает процессор так, как будто DPC-вызова не существует, используя только что рассмотренные принципы выбора.

Такое поведение нацелено на то, чтобы повысить производительность и масштабируемость серверных систем, использующих Hyper-V, хотя она может повысить производительность на любой сильно загруженной системе. По мере накопления системных таймеров, из-за того что большинство драйверов не создают родственных связей между своими DPC-вызовами, центральный процессор 0 становится

все более и более загруженным выполнением кода истечения времени таймера. Это повышает время отклика и может даже стать причиной длительных задержек выполнения DPC-вызовов или их утраты. Кроме того, истечение времени таймера может начать конкурировать с DPC-таймером, обычно связанным с обработкой прерывания, инициированного драйвером, например кодом сетевого пакета, что приводит к замедлению работы всей системы. Этот процесс усугубляется в сценарии Hyper-V, где центральный процессор 0 должен обрабатывать таймеры и DPC-вызовы, связанные с потенциальным множеством виртуальных машин, у каждой из которых есть свои таймеры и связанные с ними устройства.

Путем распределения таймеров между процессорами, как показано на рис. 3.10, загрузка каждого процессора, связанная с истечением времени таймера, полностью распределяется между незапаркованными логическими процессорами. На 32-разрядных системах объект таймера сохраняет номер связанного с ним процессора в заголовке диспетчера, а на 64-разрядныйх системах этот номер сохраняется в самом объекте.

ПРИМЕЧАНИЕ

Это поведение управляется переменной ядра KiDistributeTimers, которая инициализируется на основе параметра реестра, имеющего разное значение для установки сервера и клиента. Это поведение может быть настроено по-разному путем изменения или создания значения DistributeTimers, отличного от его исходного, основанного на SKU значения в параметре HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\kernel.

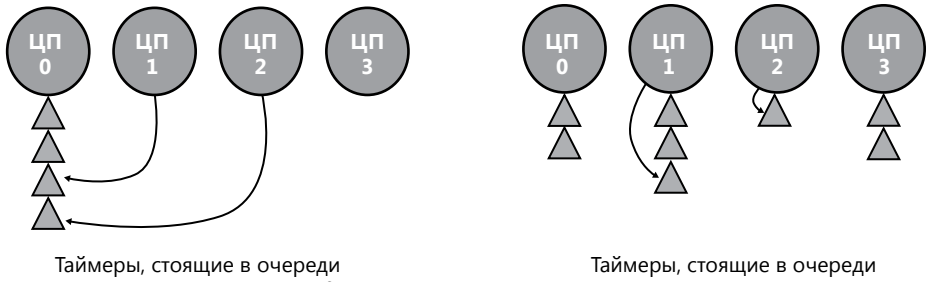


Рис. 3.10. Поведение таймеров,стоящих в очереди

ЭКСПЕРИМЕНТ: ВЫВОД СПИСКА СИСТЕМНЫХ ТАЙМЕРОВ

Чтобы вывести дамп всех, на данный момент зарегистрированных в системе таймеров, и получить информацию о DPC-вызовах, связанных с каждым таймером (если таковые имеются), можно воспользоваться отладчиком ядра. В качестве примера посмотрите на следующий вывод.

```
[lkd> !timer
Dump system timers
Interrupt time: 61876995 000003df [ 4/ 5/2010 18:58:09.189]
List Timer    Interrupt Low/High     Fire Time              DPC/thread
PROCESSOR 0 (nt!_KTIMER_TABLE fffff80001bfd080)
  5 fffffa8003099810   627684ac 000003df [ 4/ 5/2010 18:58:10.756]
NDIS!ndisMTimerObjectDpc (DPC @ fffffa8003099850)
```

```
13 fffffa8003027278   272dde78 000004cf [ 4/ 6/2010 23:34:30.510]  NDIS!ndisMWakeUpDpcX
     (DPC @ fffffa80030272b8)
   fffffa8003029278   272e0588 000004cf [ 4/ 6/2010 23:34:30.511]  NDIS!ndisMWakeUpDpcX
     (DPC @ fffffa80030292b8)
   fffffa8003025278   272e0588 000004cf [ 4/ 6/2010 23:34:30.511]  NDIS!ndisMWakeUpDpcX
     (DPC @ fffffa80030252b8)
   fffffa8003023278   272e2c99 000004cf [ 4/ 6/2010 23:34:30.512]  NDIS!ndisMWakeUpDpcX
     (DPC @ fffffa80030232b8)
 16 fffffa8006096c20    6c1613a6 000003df [ 4/ 5/2010 18:58:26.901]   thread
     fffffa8006096b60
 19 fffff80001c85c40   64f9aeb5 000003df [ 4/ 5/2010 18:58:14.971]
nt!CmpLazyFlushDpcRoutine (DPC @ fffff80001c85c00)
31 fffffa8002c43660 P dc527b9b 000003e8 [ 4/ 5/2010 20:06:00.673]
intelppm!LongCapTraceDpc (DPC @ fffffa8002c436a0)
 40 fffff80001c86f60   62ca1080 000003df [ 4/ 5/2010 18:58:11.304]  nt!CcScanDpc (DPC
      @ fffff80001c86f20)
    fffff88004039710   62ca1080 000003df [ 4/ 5/2010 18:58:11.304]
luafv!ScavengerTimerRoutine (DPC @ fffff88004039750)
...
252 fffffa800458ed50   62619a91 000003df [ 4/ 5/2010 18:58:10.619]  netbt!TimerExpiry
      (DPC @ fffffa800458ed10)
   fffffa8004599b60   fe2fc6ce 000003e0 [ 4/ 5/2010 19:09:41.514]  netbt!TimerExpiry
      (DPC @ fffffa8004599b20)
PROCESSOR 1 (nt!_KTIMER_TABLE fffff880009ba380)
  0 fffffa8004ec9700   626be121 000003df [ 4/ 5/2010 18:58:10.686]  thread
fffffa80027f3060
    fffff80001c84dd0 P 70b3f446 000003df [ 4/ 5/2010 18:58:34.647]
nt!IopIrpStackProfilerTimer (DPC @ fffff80001c84e10)
11 fffffa8005c26cd0   62859842 000003df [ 4/ 5/2010 18:58:10.855]  afd!AfdTimeoutPoll
      (DPC @ fffffa8005c26c90)
    fffffa8002ce8160   6e6c45f4 000003df [ 4/ 5/2010 18:58:30.822]  thread
fffffa80053c2b60
    fffffa8004fdb3d0   77f0c2cb 000003df [ 4/ 5/2010 18:58:46.789]  thread
fffffa8004f4bb60
 13 fffffa8005051c20   60713a93 800003df [         NEVER         ]  thread
fffffa8005051b60
 15 fffffa8005ede120   77f9fb8c 000003df [ 4/ 5/2010 18:58:46.850]  thread
fffffa8005ede060
 20 fffffa8004f40ef0   629a3748 000003df [ 4/ 5/2010 18:58:10.990]  thread
fffffa8004f4bb60
 22 fffffa8005195120   6500ec7a 000003df [ 4/ 5/2010 18:58:15.019]  thread
fffffa8005195060
 28 fffffa8004760e20   62ad4e07 000003df [ 4/ 5/2010 18:58:11.115]  btaudio (DPC
      @  fffffa8004760e60)+12d10
 31 fffffa8002c40660 P dc527b9b 000003e8 [ 4/ 5/2010 20:06:00.673]
intelppm!LongCapTraceDpc (DPC @ fffffa8002c406a0)
 ...
232 fffff80001c85040 P 62317a00 000003df [ 4/ 5/2010 18:58:10.304]  nt!IopTimerDispatch
      (DPC @ fffff80001c85080)
```

продолжение ⇩

```
    fffff80001c26fc0 P 6493d400 000003df [ 4/ 5/2010 18:58:14.304]
nt!EtwpAdjustBuffersDpcRoutine (DPC @ fffff80001c26f80)
235 fffffa80047471a8   6238ba5c 000003df [ 4/ 5/2010 18:58:10.351]  stwrt64 (DPC
      @ fffffa80047471e8)+67d4
242  fffff880023ae480    11228580  000003e1  [  4/  5/2010  19:10:13.304]
       dfsc!DfscTimerDispatch
(DPC @ fffff880023ae4c0)
245 fffff800020156b8 P 72fb2569 000003df [ 4/ 5/2010 18:58:38.469]
hal!HalpCmcDeferredRoutine (DPC @ fffff800020156f8)
248 fffffa80029ee460 P 62578455 000003df [ 4/ 5/2010 18:58:10.553]
ataport!IdePortTickHandler (DPC @ fffffa80029ee4a0)
    fffffa8002776460 P 62578455 000003df [ 4/ 5/2010 18:58:10.553]
ataport!IdePortTickHandler (DPC @ fffffa80027764a0)
      fffff88001678500    fe2f836f  000003e0  [  4/  5/2010  19:09:41.512]
        cng!seedFileDpcRoutine
(DPC @ fffff880016784c0)
    fffff80001c25b80   885e52b3 0064a048 [12/31/2099 23:00:00.008]
nt!ExpCenturyDpcRoutine (DPC @ fffff80001c25bc0)
Total Timers: 254, Maximum List: 8
```

В данном примере есть несколько драйверных таймеров с коротким сроком истечения времени, которые связаны с драйверами Ndis.sys и Afd.sys (оба этих драйвера имеют отношение к сети), а также с драйверами аудиосистемы, Bluetooth, и ATA/IDE. Есть также фоновые, вспомогательные таймеры с истечением времени, например связанные с управлением электропитанием, ETW, сбрасыванием на диск системного реестра и с виртуализацией управления учетными записями пользователей (Users Account Control, UAC). Кроме этого существует около десятка таймеров, не имеющих связанных с ними DPC-вызовов — чаще всего это свидетельствует о том, что эти таймеры пользовательского режима или режима ядра используются для диспетчеризации ожиданий. Чтобы проверить это, можно воспользоваться командой !thread в отношении указателей потоков. И наконец, есть три интересных таймера, которые всегда присутствуют на системе Windows, это таймер, проверяющий изменение часового пояса при переходе на летнее время, таймер, проверяющий наступление нового года, и таймер, проверяющий вхождение в следующее столетие. Их можно легко распознать по довольно продолжительному сроку истечения времени, если только не выполнять данный эксперимент накануне одного из этих событий. ■

Интеллектуальное распределение обработки таймерного такта

На рис. 3.9, где изображены процессоры, выполняющие ISR-процедуру обработки прерывания от часов и обрабатывающие истечение времени таймеров, показано, что процессор 1 привлекается несколько раз (сплошные стрелки), даже при отсутствии связанных с ним таймеров с истечением времени (пунктирные стрелки). Хотя при работе процессора 1 такое поведение и требуется (для обновления времени выполнения потока или процесса и планирования состояния), что если процессор 1 простаивает (и не должен заниматься истечением времени таймеров)? Зачем ему по-прежнему обрабатывать прерывания от часов? Как уже ранее

говорилось, для обновления всеобщего системного времени и такта системных часов требуется только одна дополнительная работа. В качестве хронометра вполне достаточно назначить всего лишь один процессор (в данном случае процессор 0) и дать возможность другим процессорам оставаться в спящем состоянии. Если они будут разбужены, любые корректировки, связанные со временем, могут быть выполнены путем ресинхронизации с процессором 0.

Фактически Windows так и делает, реализуя данный принцип (который называется интеллигентным распределением обработки таймерного такта), и на рис. 3.11 показаны состояния процессоров при выполнении сценария, по которому процессор 1 находится в спящем состоянии (в отличие от прежней картины, где мы предполагали, что он выполняет какой-нибудь код). Как видите, процессор 1 пробуждается только 5 раз для обработки своих таймеров с истечением времени, что создает намного более широкий разрыв (спящий период). Ядро использует переменную KiPendingTimer, которая содержит массив из структур масок сходства, показывающих, какие логические процессоры нуждаются в получении тактового интервала для заданного исполнителя таймера (интервала такта часов). Затем оно может соответствующим образом запрограммировать контроллер прерываний, а также определить, какому процессору оно будет отправлять IPI для инициирования обработки таймеров.

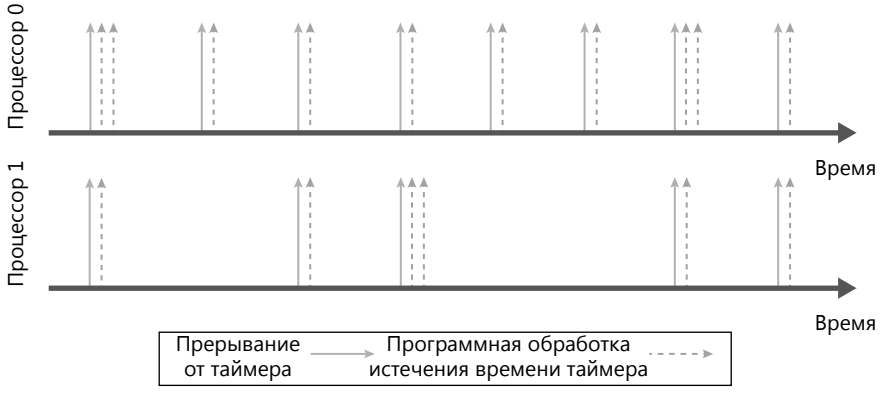


Рис. 3.11. Интеллигентное распределение обработки таймерного такта применительно к процессору 1

Оставление как можно больших промежутков играет весьма важную роль, если взять в расчет способ управления электропитанием, применяемый в процессорах: как только процессор обнаружит, что нагрузка постоянно уменьшается, он уменьшает свою потребляемую мощность (P-состояние), пока в конечном итоге не войдет в состояние простоя. У процессора есть возможность выборочно выключать свои составные части и все глубже входить в состояния простоя или спячки, например, он может выключать блоки кэш-памяти. Но если процессору снова придется пробудиться, на то чтобы включиться, будет затрачена энергия и время; поэтому разработчики процессоров, будут идти на риск, связанный с вводом этих состояний простоя и спячки (C-состояний), только если время, проводимое в этих состояниях, превосходит время и энергозатраты, затрачиваемые на вход в эти состояния и выход из них. Понятно, что нет никакого смысла

потратить 10 мс на вход в спящее состояние, которое продлится всего 1 мс. Мешая прерываниям от часов будить без надобности процессоры (из-за таймеров), можно добиться их более глубокого погружения в C-состояния и более продолжительного пребывания в этих состояниях.

Объединение таймеров

Хотя сведение к минимуму прерываний от часов для процессоров, спящих в периоды отсутствия обслуживания истечения времени таймеров, дает большой прирост более продолжительных интервалов C-состояния, при величине кванта времени в 15 мс многие таймеры, скорее всего, выстроятся в очередь к любому заданному исполнителю, и срок их действия будет истекать довольно часто, даже если будет задействован только процессор 0. Сокращение количества программной работы по обслуживанию истечения времени таймеров поможет не только сократить задержки (будет требоваться меньше работы на уровне DISPATCH_LEVEL), но и даст возможность другим процессорам еще дольше находиться в состоянии спячки (процессоры просыпаются только для обслуживания таймеров с истекшими сроками, а уменьшение количества истечений времени таймеров приводит к более длительным спящим периодам). По правде говоря, на спящее состояние реально влияет не столько количество таймеров с истечением времени (оно влияет на задержку), сколько периодичность истечения времени этих таймеров — 6 таймеров, время которых истекает по одному и тому же исполнителю предпочтительнее 6-ти таймеров, время которых истекает по 6-ти разным исполнителям. Поэтому для полной оптимизации продолжительности времени простоя ядру нужно задействовать механизм объединения (coalescing mechanism) для объединения отдельных исполнителей таймеров в единственный исполнитель с множественными истечениями времени.

Объединение таймеров работает на предположении, что большинство драйверов и приложений пользовательского режима не особо заботятся о точном периоде срабатывания своих таймеров (за исключением, к примеру, мультимедийных приложений). Эта область «не особо заботящихся» становится шире по мере роста исходного периода таймера — приложение, пробуждающееся каждые 30 с, не будет, наверное, возражать против пробуждения вместо этого каждые 31 или 29 с, а драйвер, осуществляющий опрос каждую секунду, сможет, наверное, опрашивать устройство каждую секунду плюс-минус 50 мс, не имея при этом особых проблем. Важной гарантией, от которой зависит большинство периодических таймеров, является тот факт, что период их сигналов остается постоянным в пределах определенного диапазона. К примеру, если настройка таймера была изменена на выдачу сигнала каждую секунду плюс 50 мс, он продолжает постоянно выдавать сигнал в пределах этого диапазона, не выдавая его иногда через 2, а иногда и через полсекунды. Но даже при этом не все таймеры готовы к объединению в более грубые группы детализации, поэтому Windows включает этот механизм только для таймеров, которые сами себя пометили, как имеющие такую возможность либо через API-функцию ядра KeSetCoalescableTimer, либо через ее копию пользовательского режима SetWaitableTimerEx.

С помощью этих API-функций разработчики драйверов и приложений имеют возможность предоставить ядру максимум допустимых отклонений (или

приемлемых отсрочек), которые выдержит их таймер, что определяется как максимальное количество времени за пределами запрошенного периода, при котором таймер все еще будет правильно работать[1]. Рекомендуемое минимальное допустимое отклонение составляет 32 мс, что соответствует примерно двойному 15,6-миллисекундному такту часов — любое меньшее значение не приведет к объединению, поскольку таймер с истечением срока не возможно даже будет переместить с одного такта часов на другой. Независимо от указанного допустимого отклонения Windows приводит таймер к одному из четырех наиболее предпочтительных интервалов объединения: 1 с, 250 мс, 100 мс или 50 мс.

Когда допустимая задержка устанавливается для периодического таймера, Windows использует процесс, называемый *смещением* (shifting). Смещение заставляет таймер дрейфовать между периодами, пока он не будет выровнен по наиболее оптимальному кратному периоду интервала в пределах предпочтительного объединяющего интервала, связанного с заданным допустимым отклонением (который затем кодируется в заголовке диспетчера). Для абсолютных таймеров список предпочтительных объединительных интервалов сканируется, и предпочтительное время истечения срока генерируется на основе ближайших допустимых объединительных интервалов к максимально допустимым отклонениям, указанным вызывающей программой. Такое поведение означает, что абсолютные таймеры всегда вытесняются как можно дальше от точки реального истечения времени, чтобы рассредоточить таймеры как можно дальше и создавать более продолжительные времена спячки процессоров.

Теперь, зная о возможности объединения таймеров, вернемся к рис. 3.11 и предположим, что для всех таймеров были указаны допустимые отклонения и тем самым указана допустимость объединения. По одному из сценариев Windows может принять решение по объединению таймеров, как показано на рис. 3.12. Обратите внимание, что теперь процессор 1 получает всего лишь три прерывания от часов, что существенно увеличивает периоды спячки, и способствует достижению низшего C-состояния. Кроме того, сокращается объем работы, относящейся к некоторым прерываниям от часов на процессоре 0, что, возможно, устраняет задержки, требуемые для снижения уровня до DISPATCH_LEVEL при каждом прерывании от часов.

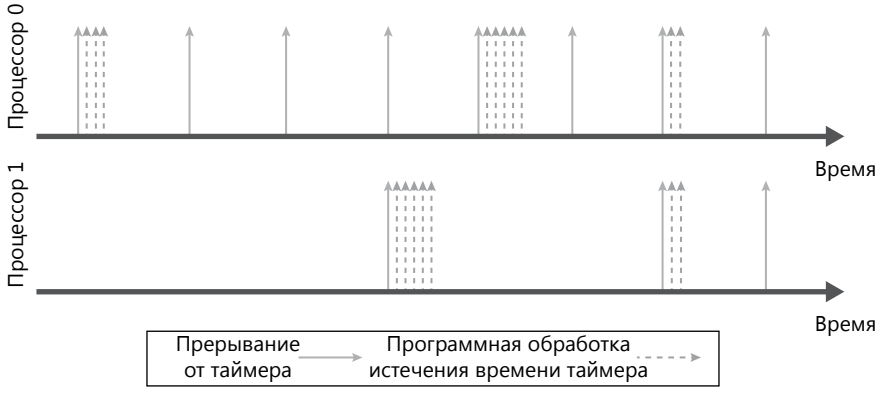


Рис. 3.12. Объединение таймеров

[1] В предыдущем примере у односекундного таймера имеется допустимое отклонение в 50 мс.

Диспетчеризация исключений

В отличие от прерываний, которые могут происходить в любое время, исключения являются условиями, являющимися непосредственными результатами выполнения запущенной программы. В Windows используется средство, известное как *структурная обработка исключения* (structured exception handling), которое позволяет приложениям получить управление при возникновении исключения. Затем приложение может исправить ситуацию и вернуться к месту возникновения исключения, вернуть назад указатель стека (завершая тем самым выполнение подпрограммы, выдавшей исключение) или объявить системе, что исключение не распознано и система должна продолжить поиск обработчика исключения, который может справиться с исключением. В данном разделе предполагается, что вы знакомы с основными понятиями, положенными в основу структурной обработки исключений Windows. Если это не так, то перед тем как продолжить чтение вам, нужно прочитать обзор справочной документации по Windows API в Windows SDK или главы с 23 по 25 в книге Джеффри Рихтера (Jeffrey Richter) и Кристофера Назара (Christophe Nasarre) «Windows via C/C++». Следует иметь в виду, что, несмотря на доступность обработки исключений средствами расширений языка программирования (например, конструкции __try в Microsoft Visual C++), данный механизм является системным и, следовательно, не имеет отношения к какому-то определенному языку.

На процессорах x86 и x64 все исключения имеют предопределенные номера прерываний, которые напрямую соотносятся с записью в IDT-таблице, указывающей на обработчик системного прерывания для конкретного исключения. В табл. 3.6 показывается исключения, определенные для x86-системы и заданные для них номера прерываний. Поскольку, как уже говорилось, первые записи в IDT-таблице используются для исключений, аппаратные прерывания назначаются тем записям, которые следуют в этой таблице позже.

Все исключения, кроме самых простых, разрешаемых с помощью обработчика системных прерываний, обслуживаются модулем ядра, который называется *диспетчером исключений*. Задачей этого диспетчера является поиск подходящего обработчика исключения. К исключениям, независящим от архитектуры и определяемым ядром, можно отнести нарушения доступа к памяти, целочисленные деления на нуль, целочисленные переполнения, исключения при работе с числами с плавающей точкой и контрольные точки отладчика. Для получения полного перечня исключений, независящих от архитектуры, нужно обратиться к справочной документации по Windows SDK.

Ядро перехватывает и обрабатывает некоторые из этих исключений прозрачно для пользовательских программ. Например, встреча контрольной точки при выполнении отлаживаемой программы приводит к выдаче исключения, которое ядро обрабатывает путем вызова отладчика. Ядро обрабатывает некоторые другие исключения, возвращая вызывающей программе код неудачного состояния.

Некоторые исключения могут передаваться в нетронутом виде назад в пользовательский режим. Например, некоторые виды ошибок обращения к памяти или арифметическое переполнение генерируют исключение, не обрабатываемое операционной системой. Для работы с этими исключениями 32-разрядные приложения

Таблица 3.6. Исключения и номера их прерываний

Номер прерывания	Исключение
0	Divide Error (Ошибка деления)
1	Debug (Single Step) (Пошаговая отладка)
2	Non-Maskable Interrupt (NMI) (Немаскируемое прерывание)
3	Breakpoint (Контрольная точка)
4	Overflow (Переполнение)
5	Bounds Check (Проверка выхода за границы)
6	Invalid Opcode (Недопустимый код операции)
7	NPX Not Available (NPX недоступен)
8	Двойная ошибка (Double Fault)
9	NPX Segment Overrun (Выход за пределы сегмента NPX)
10	Invalid Task State Segment (TSS) (Неверный сегмент состояния задачи)
11	Segment Not Present (Сегмент отсутствует)
12	Stack Fault (Ошибка стека)
13	General Protection (Общее нарушение защиты)
14	Page Fault (Ошибка обращения к странице)
15	Intel Reserved (Зарезервировано компанией Intel)
16	Floating Point (Ошибка операции с плавающей точкой)
17	Alignment Check (Ошибка проверки выравнивания)
18	Machine Check (Ошибка проверки машины)
19	SIMD Floating Point (Ошибка операции с плавающей точкой в SIMD-архитектуре)

могут устанавливать *фреймовые обработчики исключений*. Понятие *фреймовые* относится к обработчикам исключений, связанным с активацией конкретной процедуры. При вызове процедуры в стек помещается *стековый фрейм*, представляющий ее активацию. Стековый фрейм может иметь связанный с ним один или несколько обработчиков исключений, каждый из которых защищает конкретный блок кода в исходной программе. При выдаче исключения ядро осуществляет поиск обработчика исключения, связанного с текущим стековым фреймом. Если таковой отсутствует, ядро ищет обработчик исключения, связанный с предыдущим стековым фреймом, и так далее, до тех пор, пока не будет найден фреймовый обработчик исключений. Если обработчик исключений не найден, ядро вызывает свои собственные обработчики исключений, используемые по умолчанию.

Для 64-разрядных исключений в структурированной обработке исключений фреймовые обработчики не используются. Вместо этого во время компиляции в образ встраивается таблица обработчиков. Ядро ищет обработчики, связанные с каждой функцией, и следует в целом тому же алгоритму, который был описан для 32-разрядного кода.

Структурированная обработка исключений активно используется в самом ядре, поэтому оно запросто может проверить, можно ли безопасно обращаться к указателям из пользовательского режима для доступа по чтению или по записи. Драйверы могут воспользоваться такой же технологией при работе с указателями, отправленными вместе с кодами управления ввода-вывода (IOCTL-кодами).

Еще один механизм обработки исключений называется *векторной обработкой исключений*. Этот метод может быть использован только приложениями пользовательского режима. Дополнительную информацию о нем можно найти в Windows SDK или в библиотеке MSDN.

При выдаче исключения, либо явной, со стороны программного обеспечения, либо неявной, инициированной оборудованием, цепочка событий начинается в ядре. Аппаратура центрального процессора передает управление обработчику системных прерываний ядра, который создает фрейм системного прерывания (так же, как при возникновении прерывания). Фрейм системного прерывания (trap frame) позволяет системе возобновить выполнение с того же места, на котором оно было остановлено, если исключение будет разрешено. Обработчик системного прерывания также создает запись исключения, в которой содержится причина исключения и другая, относящаяся к нему информация.

Если исключение возникло в режиме ядра, диспетчер исключений просто вызывает процедуру, локализующую фреймовый обработчик исключения. Поскольку необработанные исключения режима ядра считаются фатальными ошибками операционной системы, можно предположить, что диспетчер всегда находит обработчик исключения. Тем не менее некоторые системные прерывания не приводят к обработчику исключения, потому что ядро всегда считает такие ошибки фатальными. Это относится к тем ошибкам, которые могли быть вызваны только критическими сбоями во внутреннем коде ядра или серьезными несогласованностями в коде драйвера, возникающими только благодаря преднамеренным, низкоуровневым системным изменениям, за которые драйверы не должны нести ответственность. Такие фатальные ошибки приводят к сбою проверки с кодом UNEXPECTED_KERNEL_MODE_TRAP.

Если исключение выдается в пользовательском режиме, диспетчер исключений выполняет более сложные действия. Как будет показано в главе 5, у подсистемы Windows есть порт отладки (который фактически является объектом отладчика) и порт исключения для получения уведомлений пользовательского режима в процессах Windows[1]. Как показано на рис. 3.13, ядро использует эти порты в своей обработке исключений, используемой по умолчанию.

Обычными источниками исключений являются контрольные точки отладчика. Поэтому первым делом диспетчер исключений проверяет, не связан ли процесс, выдавший исключение, с процессом отладки. Если связан, диспетчер исключений отправляет сообщение объекта отладчика *объекту отладки*, связанному с процессом. Внутри себя система обращается к объекту отладки как к «порту» для совместимости с программами, которые могут зависеть от поведения в Windows 2000, где используется не объект отладки, а LPC-порт.

Если у процесса нет подключенного к нему процесса отладчика или если отладчик не обрабатывает исключение, диспетчер исключений переключается

[1] В данном случае под термином «порт» понимается LPC-объект порта, который будет рассмотрен чуть позже.

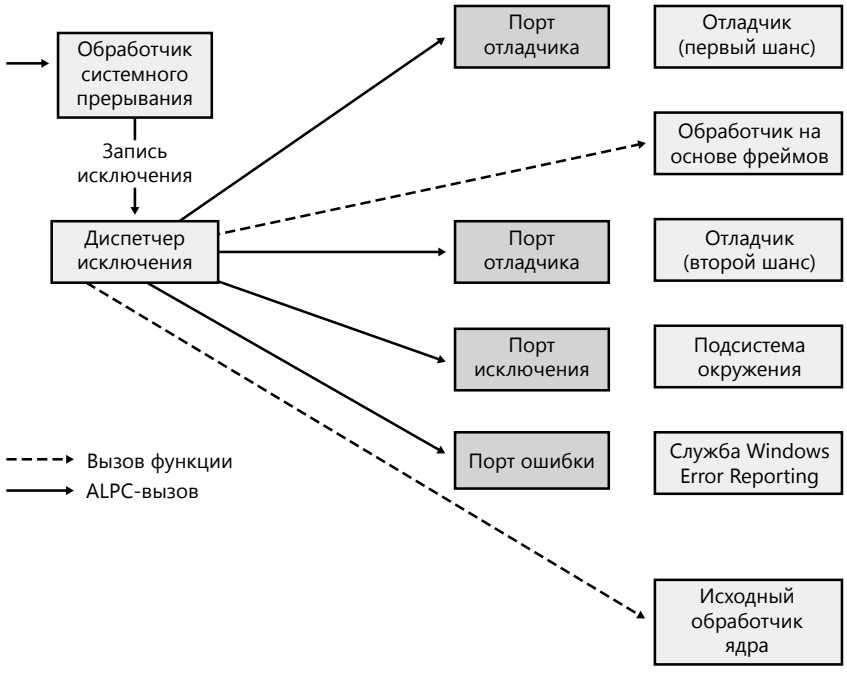


Рис. 3.13. Диспетчеризация исключения

в пользовательский режим, копирует фрейм системного прерывания в пользовательский стек, отформатированный как структура данных CONTEXT (документация по которой есть в Windows SDK), и вызывает процедуру для поиска структурированного или векторного обработчика исключения. Если таковой не будет найден или если исключение ничем не обрабатывается, диспетчер исключений переключается обратно в режим ядра и снова вызывает отладчик, чтобы дать возможность пользователю провести дополнительную отладку программы. Это называется повторным уведомлением.

Если отладчик не запущен и не найдено ни одного обработчика исключения пользовательского режима, ядро отправляет сообщение порту исключения, связанному с процессом потока. Этот порт исключения, если таковой имеется, был зарегистрирован подсистемой окружения, контролирующей этот поток. Порт исключения дает подсистеме окружения, которая предположительно прослушивает порт, возможность транслировать исключение в характерный для этого окружения сигнал или исключение. Например, когда подсистема для приложений UNIX — Subsystem for UNIX Applications — получает сообщение от ядра о том, что один из его потоков сгенерировал исключение, подсистема Subsystem for UNIX Applications отправляет сигнал в UNIX-стиле тому потоку, который стал причиной исключения. Но если ядро заходит столь далеко в обработке исключения и подсистема это исключение не обрабатывает, ядро отправляет сообщение на общесистемный порт ошибки, который подсистема времени выполнения клиент/сервер (Client/Server Run-Time Subsystem, Csrss) использует

для системы отчета об ошибках Windows Error Reporting (WER) и запускает обработчик исключений, используемый по умолчанию. Этот обработчик просто завершает тот процесс, чей поток стал причиной исключения.

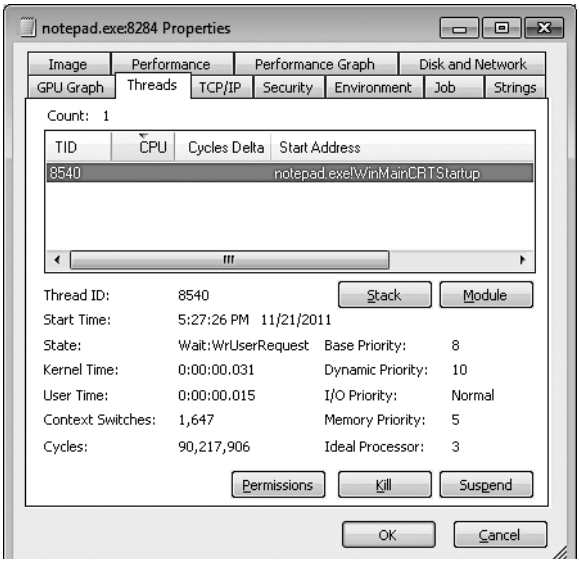
Необработанные исключения

У всех потоков Windows имеется обработчик исключений, который занимается необработанными исключениями. Этот обработчик исключений объявляется внутренней Windows-функцией запуска потока (start-of-thread). Эта функция запускается, когда пользователь создает процесс или любые дополнительные потоки. Она вызывает процедуру запуска потока, предоставляющего среду в контекстной структуре исходного потока, которая, в свою очередь, вызывает предоставляемую пользователем процедуру запуска потока, указанную в вызове CreateThread.

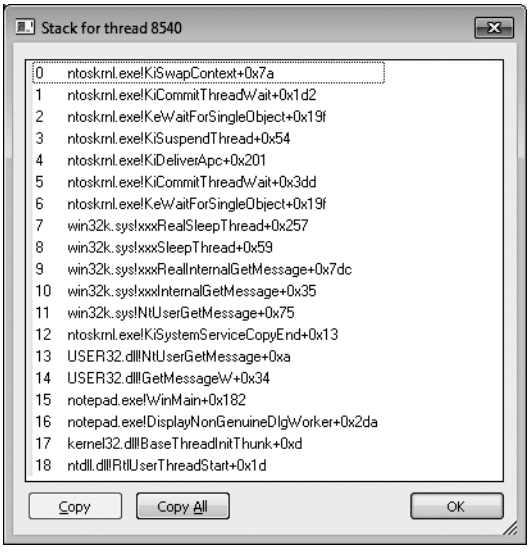
ЭКСПЕРИМЕНТ: ПРОСМОТР РЕАЛЬНОГО ПОЛЬЗОВАТЕЛЬСКОГО СТАРТОВОГО АДРЕСА ДЛЯ WINDOWS-ПОТОКОВ

Тот факт, что каждый Windows-поток начинает свое выполнение с функции, предоставляемой системой (а не пользователем), объясняет, почему стартовый адрес для потока 0 одинаков для каждого Windows-процесса, имеющегося в системе (и почему стартовые адреса для вторичных потоков также одинаковы). Для просмотра адреса функции, предоставленной пользователем, нужно воспользоваться средством Process Explorer или отладчиком ядра.

Поскольку большинство потоков в Windows-процессах берут начало в одной из предоставленных системой функциях-оболочках, Process Explorer при отображении стартового адреса потока в процессе пропускает исходный фрейм вызова, представляющий функцию-оболочку, и вместо него показывает второй фрейм в стеке. Например, обратите внимание на стартовый адрес потока того процесса, в котором запущена программа Notepad.exe.



Process Explorer при выводе стека вызовов выводит полную иерархию вызовов. Обратите внимание на следующие результаты, появляющиеся после щелчка на кнопке Stack.



В строке 18 предыдущей копии экрана показан первый фрейм стека — стартовый адрес внутренней оболочки потока. Второй фрейм (строка 17) представляет собой оболочку потока подсистемы окружения — в данном случае kernel32, поскольку вы имеете дело с приложением подсистемы Windows. Третий фрейм (строка 16) представляет собой основную точку входа в Notepad.exe.

Общий код для стартовых функций внутреннего потока имеет следующий вид:

```
VOID RtlUserThreadStart(VOID)
{
   LPVOID lpStartAddr = (R/E)AX; // Located in the initial thread context structure
   LPVOID lpvThreadParam = (R/E)BX; // Located in the initial thread context structure
    LPVOID lpWin32StartAddr;

 lpWin32StartAddr = Kernel32ThreadInitThunkFunction ? Kernel32ThreadInitThunkFunction
: lpStartAddr;
    __try
    {
        DWORD dwThreadExitCode = lpWin32StartAddr(lpvThreadParam);
        RtlExitUserThread(dwThreadExitCode);
    }
    __except(RtlpGetExceptionFilter(GetExceptionInformation()))
    {
        NtTerminateProcess(NtCurrentProcess(), GetExceptionCode());
    }
}
VOID Win32StartOfProcess(
```

продолжение ⇩

```
    LPTHREAD_START_ROUTINE lpStartAddr,
    LPVOID lpvThreadParam)
{
    lpStartAddr(lpvThreadParam);
}
```

Следует заметить, что фильтр необработанных исключений Windows вызывается в том случае, если у потока есть такое необработанное исключение. Цель данной функции заключается в предоставлении системно-независимого поведения при наличии необработанного исключения, которое выражается в запуске процесса WerFault.exe. Но в исходной конфигурации служба Windows Error Reporting, которая будет рассмотрена далее, обработает исключение, и этот фильтр необработанных исключений не будет задействован ни при каких условиях.

WerFault.exe проверяет содержимое параметра реестра HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug и убеждается в том, что процесс не находится в списке исключений. В этом параметре есть два важных значения: Auto и Debugger. Значение Auto сообщает фильтру необработанных исключений, нужно ли автоматически запускать отладчик, или же следует спросить у пользователя, что нужно делать. Установка таких средств разработки, как Microsoft Visual Studio, изменяет это значение, если оно уже установлено, на 0. (Если значение не установлено, то 0 является настройкой по умолчанию.) Значение Debugger является строкой, указывающей путь к исполняемому отладчику для его запуска в случае наличия необработанного исключения, и WerFault передает ID аварийного процесса и имя события сигналу при запуске отладчика в виде аргументов командной строки, используемой для его запуска. ■

Система Windows Error Reporting

Система отчета об ошибках Windows Error Reporting (WER) является сложным механизмом, автоматизирующим представление сбоев процессов пользовательского режима и системных сбоев режима.

Windows Error Reporting может быть настроена путем перехода в Панель управления (Control Panel) и выбора Центр поддержки (Action Center) ▸ Настройка центра поддержки (Change Action Center) ▸ Параметры отчета о неполадках (Problem Reporting Settings).

Когда фильтр необработанных исключений, упомянутый в предыдущем разделе, отлавливает такое исключение, он создает контекстную информацию (например, текущее значение регистров и стека) и открывает подключение ALPC-порта к WER-службе. Эта служба приступает к анализу состояния аварийной программы и выполняет соответствующие действия по уведомлению пользователя. Как описано ранее, во многих случаях это означает запуск программы WerFault.exe, которая выполняется с полномочиями текущего пользователя, и пока система не настроена на обратное, выводит окно сообщения, информирующее пользователя об аварии. На системах с установленным отладчиком показаны дополнительные возможности по отладке показанного процесса, что можно увидеть на рис. 3.14. При щелчке на пункте отладки (Debug) будет запущен отладчик (зарегистрированный в строке Debugger, рассмотренном ранее параметре AeDebug), чтобы подключиться к аварийному процессу.

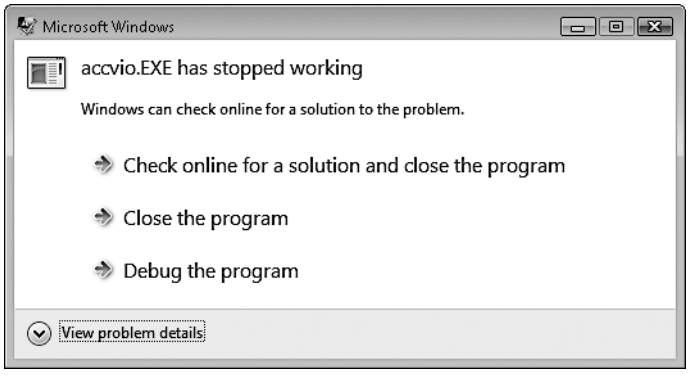


Рис. 3.14. Диалоговое окно Windows Error Reporting

На системах с исходными настройками отчет об ошибке (мини-дамп и XML-файл с различными подробностями, например с номерами версий DLL-библиотек, загруженных в процессе) отправляется на интернет-сервер Microsoft, занимающийся анализом аварийных ситуаций. Затем, когда служба уведомляется о решении для проблемы, она выводит подсказку, информирующую пользователя о его действиях, которые нужно выполнить для решения проблемы. Входящее сообщение будет также отображено в Центре поддержки. Кроме того, в Мониторе стабильности системы (Reliability Monitor) будут также показаны все экземпляры аварий приложений и системы.

ПРИМЕЧАНИЕ

WER будет активно (визуально) информировать пользователя аварийного приложения только в том случае, когда приложение имеет как минимум одно видимое интерактивное окно. В противном случае авария будет занесена в журнал, но пользователю придется вручную зайти в Центр поддержки для просмотра соответствующей записи. Такое поведение призвано избавить пользователя от путаницы, не выводя диалогового окна WER, относящегося к невидимым аварийным процессам, о которых пользователь может не знать, например о службе, выполняемой в фоновом режиме.

В окружениях, где системы не подключены к Интернету или где администратор хочет контролировать, какие именно отчеты об ошибках будут представлены Microsoft, место назначения отчета об ошибках может быть настроено на внутренний файловый сервер. Средство Microsoft System Center Desktop Error Monitoring понимает структуру каталогов, созданных Windows Error Reporting, и предоставляет администратору возможность получить избирательные отчеты об ошибках и отправить их компании Microsoft.

Если все рассмотренные операции должны были произойти в контексте сбойного потока, то есть как часть изначально установленного фильтра необработанных исключений, то иногда для слишком поврежденного потока может не получиться выполнить все эти довольно сложные шаги, и сбой может возникнуть в самом фильтре необработанных исключений. Такую «тихую смерть процесса» нельзя будет зарегистрировать, что затруднит отладку, а также выразится в невидимых сбоях в тех случаях, когда на машине не будет ни одного пользователя. Чтобы избежать подобных проблем, если сбой произошел в самом фильтре необ-

работанных исключений, имеющийся в Windows механизм WER выполняет эту работу вне аварийного потока, что позволяет зарегистрировать любую аварию процесса или потока и уведомить о ней пользователя.

WER содержит множество настраиваемых параметров, к которым пользователь может получить доступ через редактор групповой политики (Group Policy) или внося изменения в реестр вручную. В табл. 3.7 представлен список вариантов настройки WER в реестре, показано их использование и возможные значения. Эти значения находятся в подразделе HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting для настроек компьютера и в аналогичном пути в разделе HKEY_CURRENT_USER для настройки для каждого пользователя.

Таблица 3.7. Настройки WER в реестре

Настройка	Смысл	Значение
ConfigureArchive	Содержание архивных данных	1 — для параметров, 2 — для всех данных
Consent\DefaultConsent	Какие данные должны требовать согласия	1— для любых данных, 2 — только для параметров, 3 — для параметров и безопасных данных, 4 — для всех данных.
Consent\DefaultOverrideBehavior	Должен ли DefaultConsent замещать значения согласия дополнительного модуля WER	1 — для разрешения замещения
Consent\PluginName	Значение согласия для конкретного дополнительного модуля WER	То же самое, что и для DefaultConsent
CorporateWERDirectory	Каталог для общего хранилища WER	Строка, содержащая путь
CorporateWERPortNumber	Порт, используемый для общего хранилища WER	Номер порта
CorporateWERServer	Имя, используемое для общего хранилища WER	Строка, содержащая имя
CorporateWERUseAuthentication	Использование для общего хранилища WER встроенной аутентификации Windows (Windows Integrated Authentication)	1— для разрешения встроенной аутентификации
CorporateWERUseSSL	Использование для общего хранилища WER протокола защищенных сокетов (SSL)	1 — для разрешения SSL
DebugApplications	Список приложений, требующих от пользователя выбора между отладкой (Debug) и продолжением (Continue)	1— для предоставления пользователю возможности выбора

Настройка	Смысл	Значение
DisableArchive	Включен ли архив	1 — для выключения архива
Disabled	Выключена ли служба WER	1 — для выключения WER
DisableQueue	Определение необходимости постановки отчетов в очередь	1 — для выключения очереди
DontShowUI	Выключение или включение WER UI	1 — для выключения UI
DontSendAdditionalData	Предотвращение отправки дополнительных данных об аварии	1 — не отправлять
ExcludedApplications\AppName	Список приложений, исключенных из WER	Строка, содержащая список приложений
ForceQueue	Нужно ли отчеты отправлять в очередь пользователя	1 — для отправки отчетов в очередь
LocalDumps\DumpFolder	Путь, который нужно использовать для хранения дапм-файлов	Строка, содержащая путь
LocalDumps\DumpCount	Максимальное количество дапм-файлов в пути	Счетчик
LocalDumps\DumpType	Тип дампа, генерируемого при аварии	0 — для специального дампа, 1 — для мини-дампа, 2 — для полного дампа
LocalDumps\CustomDumpFlags	Для специальных дампов, указывает их специализацию	Значения, определенные в MINIDUMP_TYPE
LoggingDisabled	Включение или выключение ведения журнала	1 — для выключения ведения журнала
MaxArchiveCount	Максимальный размер архива (в файлах)	Значение в диапазоне 1–5000
MaxQueueCount	Максимальный размер очереди	Значение в диапазоне 1–500
QueuePesterInterval	Дни между запросами, чтобы пользователь мог проверить решения	Количество дней

ПРИМЕЧАНИЕ

Значения, перечисленные в параметре LocalDumps, могут также быть настроены для каждого приложения путем добавления имени приложения в пути подраздела между LocalDumps и соответствующим значением. Но они не могут быть настроены для каждого пользователя, поскольку существуют только в пути HKLM.

Как уже говорилось, для связи с аварийными процессами служба WER использует ALPC-порт. Этот механизм использует общесистемный порт ошибки, который служба WER регистрирует через функцию NtSetInformationProcess (использующую DbgkRegisterErrorPort). В результате все процессы Windows теперь имеют порт ошибки, который на самом деле является объектом ALPC-порта, зарегистрированным службой WER. Ядро, которое уведомляется об исключении в первую очередь, использует этот порт для отправки сообщения службе WER, которая затем анализирует аварийный процесс. Это означает, что даже в тяжелых случаях повреждения состояния потока WER все равно сможет получить уведомления и запустить WerFault.exe для вывода пользовательского интерфейса, не перекладывая эту работу на сам аварийный поток. Кроме того, WER сможет сгенерировать аварийный дамп процесса, и в журнал событий (Event Log) будет записано сообщение. Тем самым будут решены все проблемы тихой смерти процесса: пользователи оповещены, отладка может быть произведена, а администраторы службы могут просмотреть аварийное событие.

Диспетчеризация системных служб

Как показано на рис. 3.1, обработчики системных прерываний ядра осуществляют диспетчеризацию прерываний, исключений и вызовов системных служб. В предыдущих разделах было показано, как осуществляется обработка прерывания и исключения, а в данном разделе речь пойдет о системных службах. Диспетчер системной службы приводится в действие в результате выполнения инструкции, предназначенной для диспетчеризации этой службы. Инструкция, которую использует Windows для диспетчеризации системной службы, зависит от процессора, на которой она выполняется.

Диспетчеризация системных служб

На процессорах x86, предшествующих процессорам Pentium II, Windows использует инструкцию 0x2e (46 в десятичном выражении), которая приводит к системному прерыванию. Windows заполняет запись 46 в IDT указателем на диспетчер системных служб. (См. табл. 3.3.) Системное прерывание заставляет выполняемый поток перейти в режим ядра и войти в диспетчер системных служб. В регистр процессора EAX передается числовой аргумент, показывающий номер запрашиваемой системной службы. Регистр EDX указывает на список параметров, который вызывающая программа передает системной службе. Для возвращения в пользовательский режим диспетчер системной службы использует инструкцию iret (interrupt return — возврат из прерывания).

На процессоре x86 Pentium II и выше Windows использует инструкцию sysenter, которую компания Intel определила специально для быстродействующих диспетчеров системных служб. Для поддержки инструкции Windows сохраняет во время загрузки адрес процедуры диспетчера системных служб, находящейся в ядре, в машинно-зависимый регистр (machine-specific register, MSR), связанный с инструкцией. Выполнение инструкции приводит к переходу в режим ядра и к выполнению кода диспетчера системных служб. Номер системной службы передается в регистр процессора EAX, а регистр EDX указывает на

список аргументов вызывающей программы. Для возврата в пользовательский режим диспетчер системных служб обычно выполняет инструкцию sysexit. В некоторых случаях, например при установленном на процессоре флаге пошагового выполнения, диспетчер системных служб использует вместо нее инструкцию iret, потому что sysexit не позволяет возвращаться в пользовательский режим с измененным регистром EFLAGS, что необходимо в том случае, если инструкция sysenter была выполнена, когда флаг trap был установлен в результате трассировки, проводимой отладчиком в пользовательском режиме или при пропуске системного вызова.

ПРИМЕЧАНИЕ

Поскольку некоторые старые приложения могли быть жестко запрограммированы на использование инструкции int 0x2e для самостоятельного использования системного вызова (неподдерживаемой операции), 32-разрядная версия Windows сохраняет этот механизм готовым к использованию на системах, поддерживающих инструкцию sysenter, по-прежнему регистрируя обработчик.

В архитектуре x64 Windows использует инструкцию syscall, передавая номер системного вызова в регистре EAX, и любые параметры, вне тех четырех, в стеке.

В архитектуре IA64 Windows использует инструкцию epc (привилегированный режим ввода — Enter Privileged Mode). Первые 8 аргументов системного вызова передаются в регистрах, а остальные 8 передаются в стеке.

ЭКСПЕРИМЕНТ: ОПРЕДЕЛЕНИЕ МЕСТОПОЛОЖЕНИЯ ДИСПЕТЧЕРА СИСТЕМНЫХ СЛУЖБ

Как уже ранее говорилось, вызовы 32-разрядной системы осуществляются через прерывание, из чего следует, что обработчик должен быть зарегистрирован в IDT или через специальную инструкцию sysenter, которая во время загрузки использует для сохранения адреса обработчика регистр MSR. На некоторых 32-разрядных системах AMD Windows использует вместо нее инструкцию syscall, которая похожа на 64-разрядную инструкцию syscall. Определить местоположение соответствующей процедуры для любого метода можно следующим образом:

1. Для просмотра обработчика на 32-разрядных системах с версией системного вызова диспетчера с помощью прерывания 2E нужно набрать в отладчике ядра команду !idt 2e.

```
lkd> !idt 2e
Dumping IDT:
2e:    8208c8ee nt!KiSystemService
```

2. Для просмотра обработчика на системах с версией sysenter нужно воспользоваться командой отладчика rdmsr для чтения данных из MSR-регистра 0x176, в котором хранится адрес обработчика:

```
lkd> rdmsr 176
msr[176] = 00000000'8208c9c0
   lkd> ln 00000000'8208c9c0
(8208c9c0)   nt!KiFastCallEntry
```

При использовании 64-разрядной машины можно посмотреть на 64-разрядный диспетчер вызова служб, повторяя этот шаг, но используя вместо прежнего MSR-регистр 0xC0000082, который используется для вызова версии syscall для 64-разрядного кода. Вы увидите, что он соответствует nt!KiSystemCall64:

```
lkd> rdmsr c0000082
msr[c0000082] = fffff800'01a71ec0
lkd> ln fffff800'01a71ec0
(fffff800'01a71ec0)   nt!KiSystemCall64
```

3. Можно дизассемблировать процедуру KiSystemService или процедуру KiSystemCall64 с помощью команды u. В итоге на 32-разрядной системе вы заметите следующие инструкции:

```
nt!KiSystemService+0x7b:
8208c969 897d04          mov     dword ptr [ebp+4],edi
8208c96c fb              sti
8208c96d e9dd000000      jmp     nt!KiFastCallEntry+0x8f (8208ca4f)
```

Поскольку реальные операции диспетчеризации системных вызовов являются общими, независимо от механизма, используемого для выхода на обработчик, старый обработчик, основанный на применении прерывания, для выполнения тех же общих задач просто вызывается в середине более нового обработчика, основанного на применении инструкции sysenter. Единственные отличающиеся части обработчиков связаны с генерацией фрейма системного прерывания и установкой значений конкретных регистров. ■

Во время загрузки 32-разрядная Windows определяет тип процессора, на котором она выполняется, и устанавливает соответствующий используемый код системного вызова путем сохранения указателя на правильный код в структуре SharedUserData. Код системной службы для NtReadFile в пользовательском режиме имеет следующий вид:

```
0:000> u ntdll!NtReadFile
ntdll!ZwReadFile:
77020074 b802010000      mov     eax,102h
77020079 ba0003fe7f      mov     edx,offset SharedUserData!SystemCallStub
(7ffe0300)
7702007e ff12            call    dword ptr [edx]
77020080 c22400          ret     24h
77020083 90              nop
```

Номер системной службы — 0x102 (258 в десятичном формате), а инструкция call выполняет установленный ядром код диспетчера системной службы, чей указатель находится по адресу 0x7ffe0300. (Это соответствует элементу SystemCallStub структуры KUSER_SHARED_DATA, который начинается с адреса 0x7FFE0000.)

Поскольку следующий вывод взят из Intel Core 2 Duo, он содержит указатель на sysenter:

```
0:000> dd SharedUserData!SystemCallStub l 1
7ffe0300  77020f30
0:000> u 77020f30
```

```
ntdll!KiFastSystemCall:
77020f30 8bd4            mov     edx,esp
77020f32 0f34            sysenter
```

Так как у 64-разрядных систем есть только один механизм для осуществления системных вызовов, точки входа системной службы в Ntdll.dll, как показано здесь, напрямую используют инструкцию syscall:

```
ntdll!NtReadFile:
00000000'77f9fc60 4c8bd1          mov     r10,rcx
00000000'77f9fc63 b810200000      mov     eax,0x102
00000000'77f9fc68 0f05            syscall
00000000'77f9fc6a c3              ret
Kernel-Mode System Service Dispatching
```

Как показано на рис. 3.15, для обнаружения информации о системной службе в *таблице диспетчера системной службы* ядро использует номер системного вызова. На 32-разрядных системах эта таблица похожа на рассмотренную ранее в этой главе таблицу диспетчера прерываний, за исключением того, что каждая запись содержит указатель на системную службу, а не на процедуру обработки прерывания. На 64-разрядных системах таблица реализована несколько иначе, она содержит не указатели на системные службы, а смещения относительно самой таблицы. Этот механизм адресации лучше подходит имеющемуся в системе x64 двоичному интерфейсу прикладных программ — application binary interface (ABI) и формату кодирования инструкций.

ПРИМЕЧАНИЕ

В зависимости от используемого пакета обновлений номера системных служб могут меняться — компания Microsoft время от времени добавляет или удаляет системные службы, и номера системных служб генерируются автоматически, как часть компиляции ядра.

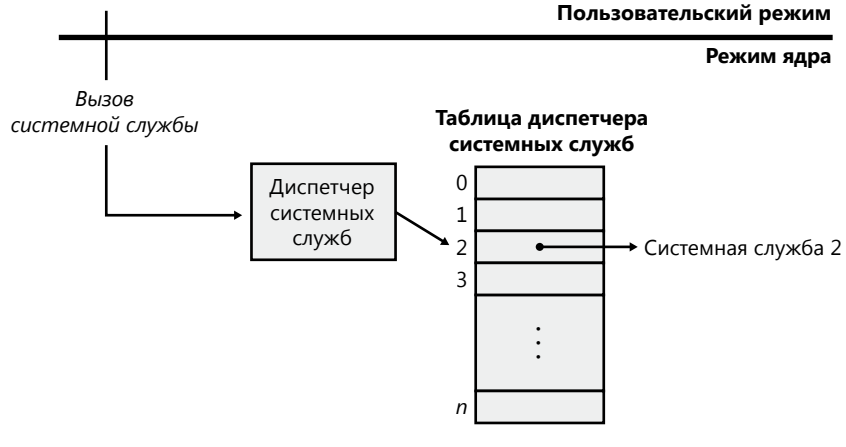


Рис. 3.15. Исключения системных служб

Диспетчер системных служб, KiSystemService, копирует аргументы вызывающей программы из стека потока пользовательского режима в свой стек режима

ядра (чтобы пользователь не смог изменить аргументы при обращении к ним ядра), а затем выполняет системную службу. Ядро получает представление о том, сколько байт стека нужно копировать, благодаря использованию второй таблицы, которая называется таблицей аргументов и является байтовым массивом (в отличие от массива указателей вроде таблицы диспетчеризации). Каждая запись дает описание количества байт для копирования. На 64-разрядных системах Windows кодирует эту информацию в саму таблицу службы посредством процесса, который называется уплотнением таблицы системных вызовов. Если аргументы, передаваемые системной службе, указывают на буферы в пользовательском пространстве, эти буферы должны быть проверены на доступность, прежде чем код режима ядра сможет копировать данные в эти буферы или из них. Эта проверка осуществляется только тогда, когда *предыдущий режим* (previous mode) потока установлен на пользовательский режим.

Предыдущий режим является значением (режим ядра или пользовательский режим), которое ядро сохраняет в потоке, когда в нем выполняется обработчик системного прерывания и идентифицируется уровень привилегий входящего исключения, системного прерывания или системного вызова. В качестве оптимизации, если системный вызов поступает от драйвера или от самого ядра, проверка и захват параметров пропускаются, и все параметры считаются указывающими на допустимые буферы режима ядра (также разрешается доступ к данным в режиме ядра).

Поскольку системные вызовы могут также осуществляться кодом, выполняемым в режиме ядра, давайте рассмотрим способ их реализации. Так как код для каждого системного вызова выполняется в режиме ядра, а вызывающая программа уже выполняется в режиме ядра, можно прийти к выводу, что какого-либо прерывания или операции sysenter не требуется: центральный процессор уже находится на нужном уровне привилегий, и драйверы, как и ядро, должны только лишь иметь возможность непосредственного вызова требуемой функции. В случае исполняющей системы именно это и происходит: ядро имеет доступ ко всем своим собственным процедурам и может просто вызвать их точно так же, как вызывает стандартные процедуры. Но внешне драйверы могут получить доступ к этим системным вызовам только тогда, когда эти вызовы экспортированы подобно другим стандартным API-функциям режима ядра. Фактически, экспортировано довольно много системных вызовов. Но такой способ доступа для драйверов не предусматривается. Вместо этого драйверы должны использовать Zw-версии этих вызовов, то есть вместо NtCreateFile они должны использовать ZwCreateFile. Эти Zw-версии должны быть также вручную экспортированы ядром, их немного, но на них имеется полная документация и поддержка.

Из-за рассмотренного ранее понятия *предыдущего режима* Zw-версии официально доступны только для драйверов. Поскольку значение предыдущего режима обновляется только при каждом создании ядром фрейма системного прерывания, в связи с простым API-вызовом оно изменяться не будет, поскольку никакого фрейма системного вызова сгенерировано не будет. При непосредственном вызове таких функций, как NtCreateFile, ядро сохраняет значение *предыдущего режима*, которое показывает, что оно относится к пользовательскому режиму, обнаруживает, что переданный адрес относится к адресу режима ядра, и не выполняет вызов, правильно полагая, что приложения пользовательского режима не должны передавать указатели режима ядра. Но на самом деле ведь это не так,

тогда как же ядро должно разобраться в правильном *предыдущем режиме*? Ответ заключается в Zw-вызовах.

Эти экспортированные API-функции на самом деле не являются простыми псевдонимами или оболочками Nt-версий. Вместо этого они являются своеобразными «батутами» для прыжка к соответствующим системным Nt-вызовам, использующим тот же самый механизм диспетчеризации системных вызовов. Вместо генерирования прерывания или использования инструкции sysenter, которые не отличались бы скоростью работы и (или) не поддерживались бы, они создают искусственный стек прерывания (стек, который центральный процессор сгенерировал бы после прерывания) и непосредственно вызывают процедуру KiSystemService, фактически имитируя прерывание центрального процессора. Обработчик выполняет те же операции, что и при поступлении этого вызова из пользовательского режима, за исключением того, что он обнаруживает фактический уровень привилегий, с которым поступил вызов, и устанавливает для *предыдущего режима* значение режима ядра (kernel). Теперь функция NtCreateFile видит, что вызов поступил из ядра, и больше не отвечает отказом. Далее показано, как выглядят батуты режима ядра на 32-разрядной и на 64-разрядной системах. Номер системного вызова выделен жирным шрифтом.

```
lkd> u nt!ZwReadFile
nt!ZwReadFile:
8207f118 b802010000      mov     eax,102h
8207f11d 8d542404        lea     edx,[esp+4]
8207f121 9c              pushfd
8207f122 6a08            push    8
8207f124 e8c5d70000      call    nt!KiSystemService (8208c8ee)
8207f129 c22400          ret     24h
lkd> uf nt!ZwReadFile
nt!ZwReadFile:
fffff800'01a7a520 488bc4          mov     rax,rsp
fffff800'01a7a523 fa              cli
fffff800'01a7a524 4883ec10        sub     rsp,10h
fffff800'01a7a528 50              push    rax
fffff800'01a7a529 9c              pushfq
fffff800'01a7a52a 6a10            push    10h
fffff800'01a7a52c 488d05bd310000  lea     rax,[nt!KiServiceLinkage
(fffff800'01a7d6f0)]
fffff800'01a7a533 50              push    rax
fffff800'01a7a534 b803000000      mov     eax,3
fffff800'01a7a539 e902690000      jmp     nt!KiServiceInternal (fffff800'01a80e40)
```

В главе 5 будет показано, что в Windows есть две таблицы системных служб, и драйверы сторонних разработчиков не могут расширить таблицы или вставить новые для добавления своих собственных вызовов служб. На 32-разрядных версиях Windows и на версиях IA64 диспетчер системных служб определяет местоположение таблиц через указатель в структуре потоков ядра, а на x64-версиях он находит их по их глобальным адресам. Диспетчер системных служб определяет, в какой таблице содержится запрошенная служба, интерпретируя двухразрядное поле в 32-разрядном номере системной службы в качестве индекса

таблицы. Младшие 12 разрядов номера системной службы служат в качестве индекса в таблице, указанной индексом таблицы. Поля показаны на рис. 3.16.

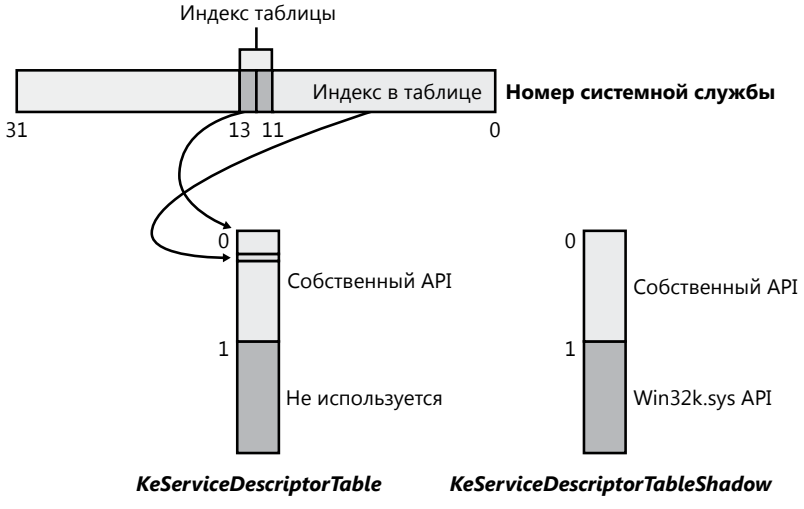


Рис. 3.16. Преобразование номера системной службы к системной службе

Таблицы дескрипторов служб

Главная таблица, используемая по умолчанию, *KeServiceDescriptorTable*, определяет основные системные службы исполняющей системы, реализованные в файле Ntosrknl.exe. Другая таблица, *KeServiceDescriptorTableShadow*, включает Windows-службы USER и GDI, реализованные в той части подсистемы Windows, которая работает в режиме ядра, Win32k.sys. На 32-разрядной версии Windows и на версии IA64, где поток Windows сразу же вызывает Windows-службу USER или службу GDI, адрес таблицы системных служб потока изменяется, чтобы указать на таблицу, которая включает Windows-службы USER и GDI. Функция KeAddSystemServiceTable позволяет Win32k.sys добавлять таблицу системных служб.

Инструкции диспетчера системных служб для служб исполняющей системы Windows находятся в системной библиотеке Ntdll.dll. Для реализации своих документированных функций DLL-библиотеки подсистем вызывают функции в Ntdll. Исключение составляют функции Windows USER и GDI, для которых инструкции диспетчера системных служб реализованы в User32.dll и Gdi32.dll — Ntdll.dll в данном случае не привлекается. Эти два обстоятельства показаны на рис. 3.17.

Как показано на рис. 3.17, Windows-функция WriteFile в Kernel32.dll импортирует и вызывает функцию WriteFile в API-MS-Win-Core-File-L1-1-0.dll, в одной из DLL-библиотек перенаправления MinWin (более подробно API-перенаправление будет рассмотрено в следующем разделе), которая, в свою очередь, вызывает функцию WriteFile в KernelBase.dll, где находится ее фактическая реализация. После проверки некоторых параметров, характерных для подсистемы, она затем вызывает функцию NtWriteFile в Ntdll.dll, которая, в свою очередь, выполняет соответствующую инструкцию, чтобы вызвать системное прерывание системной службы, передавая номер системной службы, представляющей NtWriteFile. Затем

диспетчер системных служб (функция KiSystemService в Ntoskrnl.exe) вызывает настоящую функцию NtWriteFile для обработки запроса ввода-вывода. Для функций Windows USER и GDI диспетчер системных служб вызывает функции в загружаемой части подсистемы Windows режима ядра, Win32k.sys.

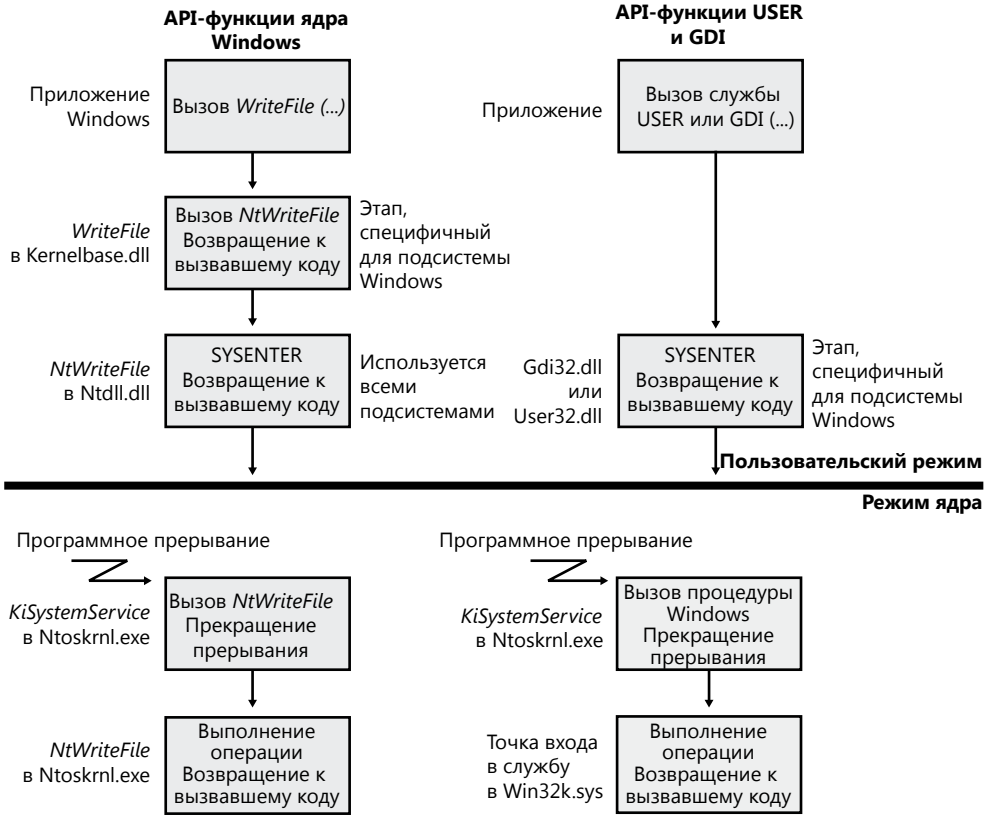


Рис. 3.17. Диспетчеризация системных служб

ЭКСПЕРИМЕНТ: ОТОБРАЖЕНИЕ НОМЕРОВ СИСТЕМНЫХ СЛУЖБ НА ФУНКЦИИ И АРГУМЕНТЫ

Такой же поиск можно продублировать с помощью ядра, работая с идентификатором системного вызова для определения, какая именно функция отвечает за его обработку и сколько аргументов она принимает

1. Обе таблицы, *KeServiceDescriptorTable* и *KeServiceDescriptorTableShadow*, направляют на один и тот же массив указателей (или смещений на 64-разрядной системе) для системных вызовов ядра, который называется *KiServiceTable*, и на один и тот же массив байтов стека, который называется *KiArgumentTable*. На 32-разрядной системе для получения дампа данных наряду с символьной информацией можно воспользоваться

командой отладчика ядра dds. Отладчик пытается сопоставить каждый указатель с символом. Частичный вывод имеет следующий вид:

```
lkd> dds KiServiceTable
820807d0  821be2e5 nt!NtAcceptConnectPort
820807d4  820659a6 nt!NtAccessCheck
820807d8  8224a953 nt!NtAccessCheckAndAuditAlarm
820807dc  820659dd nt!NtAccessCheckByType
820807e0  8224a992 nt!NtAccessCheckByTypeAndAuditAlarm
820807e4  82065a18 nt!NtAccessCheckByTypeResultList
820807e8  8224a9db nt!NtAccessCheckByTypeResultListAndAuditAlarm
820807ec  8224aa24 nt!NtAccessCheckByTypeResultListAndAuditAlarmByHandle
820807f0  822892af nt!NtAddAtom
```

2. Как уже ранее говорилось, 64-разрядная Windows создает таблицу системных вызовов по-другому и использует относительные указатели (или смещения) к системным вызовам, а не абсолютные адреса, используемые в 32-разрядной Windows. Базой указателя служит сама таблица *KiServiceTable*, поэтому выводить дамп данных в его необработанном формате придется с помощью команды dq. Пример вывода из 64-разрядной системы имеет следующий вид:

```
lkd> dq nt!KiServiceTable
fffff800'01a73b00  02f6f000'04106900 031a0105'fff72d00
```

3. Вместо вывода дампа всей таблицы вы также можете найти конкретный номер. На 32-разрядной Windows, благодаря тому, что номер каждого системного вызова является индексом в таблице, и тому, что каждый элемент занимает 4 байта, можно воспользоваться следующим вычислением: Обработчик = KiServiceTable + Номер * 4. Давайте возьмем номер 0x102, полученный во время описания кода функции-заглушки NtReadFile в Ntdll.dll.

```
lkd> ln poi(KiServiceTable + 102 * 4)
(82193023)   nt!NtReadFile
```

На 64-разрядной Windows каждое смещение может быть сопоставлено с каждой функцией с помощью команды ln, путем смещения вправо на 4 разряда (используемых согласно ранее данному описанию) и добавления оставшегося значения к базе самой таблицы *KiServiceTable*, как показано в следующем примере:

```
lkd> ln @@c++(((int*)@@(nt!KiServiceTable))[3] >> 4) + nt!KiServiceTable
(fffff800'01d9cb10)   nt!NtReadFile   |  (fffff800'01d9d24c)   nt!NtOpenFile
Exact matches:
    nt!NtReadFile = <no type information>
```

4. Поскольку драйверы, включая руткиты режима ядра, способны на 32-разрядных версиях Windows делать вставки в эту таблицу, что операционной системой не поддерживается, команду dds можно использовать для получения дампа всей таблицы и для поиска любых значений за пределами диапазона допустимых адресов ядра (dds также даст это понять, не давая возможности просматривать символ для функции). На 64-разрядной Windows таблицы системных служб отслеживаются системой за-

щиты от правки ядра — Kernel Patch Protection, которая при обнаружении изменений вводит систему в аварийное состояние. ■

ЭКСПЕРИМЕНТ: ПРОСМОТР ПРОЯВЛЕНИЯ АКТИВНОСТИ СИСТЕМНОЙ СЛУЖБЫ

Проявление активности системной службы можно отследить, наблюдая за счетчиком производительности, показывающим количество системных вызовов в секунду — Системных вызовов/с (System Calls/Sec) в объекте Система (System). Запустите Системный монитор (Performance Monitor), щелкните на пункте Системный монитор (Performance Monitor) в разделе Средства наблюдения (Monitoring Tools) и щелкните на кнопке Добавить (Add), чтобы добавить счетчик к графику. Выберите объект Система (System), выберите счетчик Системных вызовов/с (System Calls/Sec), а затем щелкните на кнопке Добавить (Add) для добавления счетчика к графику. ■

Диспетчер объектов

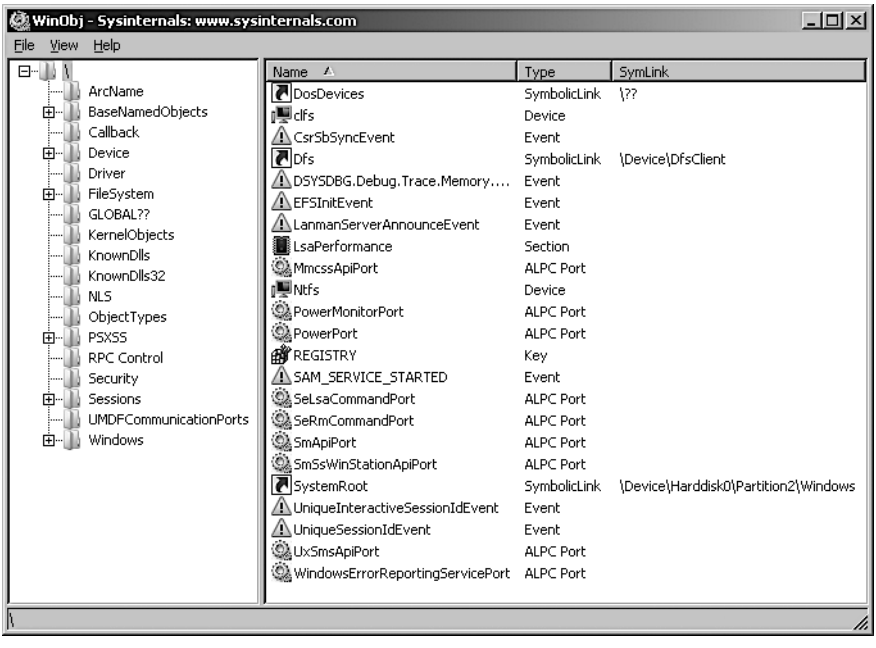
Как уже говорилось в главе 2 «Архитектура системы», для обеспечения последовательного и безопасного доступа к различным внутренним службам, реализованным в исполняющей системе, в Windows реализуется объектная модель. В этом разделе рассматривается *диспетчер объектов* Windows, компонент исполняющей системы, отвечающий за создание, удаление, защиту и отслеживание объектов. Диспетчер объектов централизует операции управления, которые в противном случае были бы разбросаны по всей операционной системе. Он был разработан для достижения целей, перечисленных далее.

ЭКСПЕРИМЕНТ: ИССЛЕДОВАНИЕ ДИСПЕТЧЕРА ОБЪЕКТОВ

В данном разделе будут представлены эксперименты, показывающие способы изучения базы данных диспетчера объектов. В этих экспериментах используются следующие средства, с которыми вы познакомитесь, если они вам еще не известны:

- WinObj (доступно на сайте Sysinternals) показывает пространство имен диспетчера внутренних объектов и информацию об объектах (например, счетчик ссылок, количество открытых дескрипторов, дескрипторов безопасности и т. д.).
- Process Explorer и Handle от Sysinternals, а также Монитор ресурсов (Resource Monitor) (представленный в главе 1) выводят открытые дескрипторы процесса.
- Команда Openfiles/query показывает открытые для процесса файловые дескрипторы, но для ее работы требуется установка глобального флага.
- Команда отладчика ядра !handle показывает открытые для процесса дескрипторы.

Средство WinObj предоставляет способ проникновения в пространство имен, поддерживаемое диспетчером объектов. (Чуть позже будет объяснено, что имена имеются не у всех объектов.) Запустите WinObj и изучите показанную ниже схему.



Как уже упоминалось, Windows-команда Openfiles/query требует установки глобального флага Windows, который называется флагом обслуживания списка объектов — maintain objects list. (Для получения более подробной информации о глобальных флагах см. далее раздел «Глобальные флаги Windows».) Если набрать команду Openfiles/Local, можно получить информацию о том, установлен этот флаг или нет. Этот флаг можно установить с помощью команды Openfiles/Local ON. В любом случае, чтобы установка возымела эффект, нужно перезапустить систему. Process Explorer, Handle и Монитор ресурсов (Resource Monitor) не требуют включения отслеживания объектов, поскольку они запрашивают все системные дескрипторы и создает список объектов, принадлежащих каждому объекту. ■

Диспетчер объектов был разработан для достижения следующих целей:

- обеспечение общего, унифицированного механизма для использования системных ресурсов;
- изоляция защиты объектов в одном месте операционной системы для обеспечения унификации и последовательности политики доступа к объектам;
- обеспечение механизма зарядки процессов на использование ими объектов, чтобы можно было наложить лимиты на использование системных ресурсов;
- учреждение схемы названия объектов, в которую могут быть легко включены существующие объекты, например устройства, файлы и каталоги файловой системы или другие независимые коллекции объектов;
- поддержка требований различных окружений операционной системы, таких как возможность наследования процессом ресурсов от родительского процесса (необходимых Windows и подсистеме Subsystem for UNIX Applications)

и возможность создания имен файлов, чувствительных к регистру букв (необходимых для подсистемы Subsystem for UNIX Applications);

- учреждение унифицированных правил для сохранения объектов (object retention) (то есть для сохранения объектов доступными до тех пор, пока все процессы не завершат их использование);
- обеспечение возможности изоляции объектов для конкретного сеанса, чтобы учесть в пространстве имен как локальные, так и глобальные объекты.

Согласно своему внутреннему устройству у Windows есть три типа объектов: объекты исполняющей системы, объекты ядра и объекты GDI/User. Объекты исполняющей системы представлены объектами, реализованными различными компонентами исполняющей системы (такими как диспетчер процессов, диспетчер памяти, подсистема ввода-вывода и т. д.). Объекты ядра представлены более простым набором, реализованным ядром Windows. Эти объекты не видимы коду пользовательского режима и создаются и используются только внутри исполняющей системы. Объекты ядра обеспечивают такие основные возможности, как синхронизация, на которых построены объекты исполняющей системы. Таким образом, как показано на рис. 3.18, многие объекты исполняющей системы содержат (инкапсулируют) один или несколько объектов ядра.

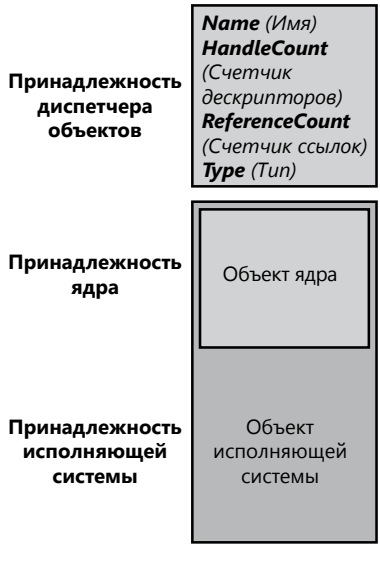


Рис. 3.18. Объекты исполняющей системы, содержащие объекты ядра

ПРИМЕЧАНИЕ

С другой стороны, объекты GDI/User принадлежат подсистеме Windows (Win32k.sys) и не взаимодействуют с ядром. Поэтому в сферу рассмотрения данной книги они не входят, но получить дополнительную информацию о них можно из Windows SDK.

Подробности структуры объектов ядра и того, как они используются для реализации синхронизации, будут рассмотрены чуть позже. Вся оставшаяся

часть этого раздела будет сфокусирована на том, как работает диспетчер объектов, и на структуре объектов исполняющей системы, дескрипторах и таблицах дескрипторов, в ней также будет дано краткое описание использования объектов в реализации в Windows проверки безопасности доступа, а более основательно эта тема будет раскрыта в главе 6.

Объекты исполняющей системы

Каждая подсистема среды окружения Windows создает для своих приложений различный образ операционной системы. Объекты исполняющей системы и службы объектов являются элементами, использующимися подсистемами среды окружения для создания их собственных версий объектов и других ресурсов.

Объекты исполняющей системы обычно создаются либо подсистемой среды окружения от имени пользовательского приложения, либо различными компонентами операционной системы в рамках их обычного функционирования. Например, для создания файла приложение Windows вызывает Windows-функцию CreateFileW, реализованную в DLL-библиотеке подсистемы Windows Kernelbase.dll. После проверки и инициализации функция CreateFileW, в свою очередь, для создания объекта файла исполняющей системы вызывает исходную Windows-службу NtCreateFile.

Набор объектов подсистемы среды окружения, предоставляемый ее приложениям, может быть больше или меньше набора, предоставляемого исполняющей системой. Подсистема Windows использует объекты исполняющей системы для экспорта своего собственного набора объектов, многие из которых напрямую связаны с объектами исполняющей системы. Например, мьютексы и семафоры Windows непосредственно основаны на объектах исполняющей системы (которые, в свою очередь, основаны на соответствующих объектах ядра). Кроме того, подсистема Windows предоставляет именованные каналы и почтовые слоты, ресурсы, основанные на объектах файлов исполняющей системы. Некоторые подсистемы, такие как Subsystem for UNIX Applications, вообще не поддерживают объекты как таковые. Subsystem for UNIX Applications использует объекты исполняющей системы и службы в качестве основы для представления процессов, каналов и других ресурсов в стиле UNIX для своих приложений.

ПРИМЕЧАНИЕ

В исполняющей системе реализовано всего 4242 типа объектов. Многие из тих объектов предназначены только для использования теми компонентами исполняющей системы, которые их определили, и получить непосредственный доступ к ним из функций Windows API невозможно. В качестве примера таких объектов можно привести Driver, Device, и EventPair.

В табл. 3.8 перечислены первичные объекты, предоставляемые исполняющей системой, и дано краткое описание того, что они представляют. Более подробную информацию об объектах исполняющей системы можно найти в главах, описывающих связанные с ними компоненты этой системы, а для объектов исполняющей системы, непосредственно экспортируемых в Windows, — в справочной докумен-

тации по Windows API. Полный список типов объектов можно получить, после запуска Winobj с привилегированными правами и перехода в каталог ObjectTypes.

Таблица 3.8. Объекты исполняющей системы, видимые функциям Windows API

Тип объекта	Что он представляет
Process (Процесс)	Виртуальное адресное пространство и управляющую информацию, необходимую для выполнения набора объектов типа «поток»
Thread (Поток)	Исполняемая категория внутри процесса
Job (Задание)	Коллекция процессов, управляемых как единое целое в рамках задания
Section (Раздел)	Область разделяемой памяти (известная в Windows как проекция файла)
File (Файл)	Экземпляр открытого файла или устройства ввода-вывода
Token (Маркер)	Профиль безопасности (идентификатор безопасности, права пользователя и т.д.) процесса или потока
Event (Событие)	Объект, имеющий постоянное состояние (о котором поступил или не поступил сигнал), который может использоваться для синхронизации или уведомления
Semaphore (Семафор)	Счетчик, ограничивающий доступ к ресурсу путем разрешения доступа к этому ресурсу, защищенному семафором, вполне определенному максимальному количеству потоков
Mutex (Мьютекс)	Механизм синхронизации, используемый для последовательного доступа к ресурсу
Timer (Таймер)	Механизм уведомления потока об истечении конкретного периода времени
IoCompletion (Завершение ввода-вывода)	Метод для потоков по постановке в очередь и извлечении из нее уведомлений о завершении операций ввода-вывода (известный в Windows API как порт завершения ввода-вывода)
Key (Раздел реестра)	Механизм ссылки на данные реестра. Хотя разделы появляются в пространстве имен диспетчера объектов, они управляются диспетчером конфигурации точно так же, как файловые объекты управляются драйверами файловой системы. С объектом раздела (key) связано от нуля до нескольких значений раздела, эти значения содержат данные о разделе
Directory (Каталог)	Виртуальный каталог в пространстве имен диспетчера объектов, отвечающий за содержание других объектов или каталогов объектов
TpWorkerFactory	Коллекция потоков, назначенных для выполнения конкретного набора задач. Ядро может управлять количеством рабочих элементов, которые будут выполняться по очереди, тем, сколько именно потоков будут отвечать за работу, а также динамическим созданием и завершением рабочих потоков, исходя из конкретных ограничений, устанавливаемых вызывающей программой. Windows показывает рабочий производственный объект (worker factory object) через пулы потоков

продолжение ⇩

Таблица 3.8 *(продолжение)*

Тип объекта	Что он представляет
TmRm (Диспетчер ресурсов), TmTx (Транзакция), TmTm (Диспетчер транзакций), TmEn (Включение в список)	Объекты, используемые диспетчером транзакций ядра (Kernel Transaction Manager, KTM) для различных транзакций и (или) включений в списки в качестве части диспетчера ресурсов или диспетчера транзакций. Объекты могут создаваться через API-функции CreateTransactionManager, CreateResourceManager, CreateTransaction и CreateEnlistment
WindowStation (Станция окна)	Объект, содержащий буфер обмена, набор глобальных атомов и группу объектов типа Рабочий стол
Desktop (Рабочий стол)	Объект, содержащийся внутри объекта станции окна (window station). Рабочий стол имеет логическую поверхность дисплея и содержит окна, меню и связи
PowerRequest	Объект, связанный с выполняемым потоком, который помимо всего прочего является вызовом функции SetThreadExecutionState для запроса заданного изменения режима электропитания, такого как блокировка перехода в спящий режим (например, при воспроизведении какого-нибудь фильма)
EtwConsumer	Представляет собой подключенный ETW-потребитель режима реального времени, зарегистрированный с помощью API-функции StartTrace (и способный вызвать функцию ProcessTrace для получения событий в очереди объектов)
EtwRegistration	Представляет собой объект регистрации, связанный с ETW-провайдером пользовательского режима (или режима ядра), который зарегистрирован с помощью API-функции EventRegister

ПРИМЕЧАНИЕ

Поскольку изначально предполагалось, что Windows NT будет поддерживать операционную систему OS/2, мьютекс должен был быть совместим с существующей конструкцией объектов взаимного исключения OS/2, то есть иметь конструкцию, от которой требовалось, чтобы поток мог отказаться от объекта, оставив его недоступным. Поскольку подобное поведение для такого объекта считалось необычным, был создан еще один объект ядра — мутант. Со временем от поддержки OS/2 отказались, и объект стал использоваться подсистемой Windows 32 под названием мьютекс (но при этом он сохранил внутреннее имя мутант).

Структура объекта

Как показано на рис. 3.19, у каждого объекта есть заголовок и тело объекта. Диспетчер объектов управляет заголовками объектов, а телами объектов управляют компоненты исполняющей системы, являющиеся владельцами тех типов объектов, которые ими созданы. В каждом заголовке объекта содержится также индекс на специальный объект, называемый *объектом типа* (type object), в котором содержится информация, общая для каждого экземпляра объекта. Кроме того, существует до пяти дополнительных подзаголовков: информационный

заголовок названия, информационный заголовок квоты, информационный заголовок процесса, информационный заголовок дескриптора и информационный заголовок создателя.

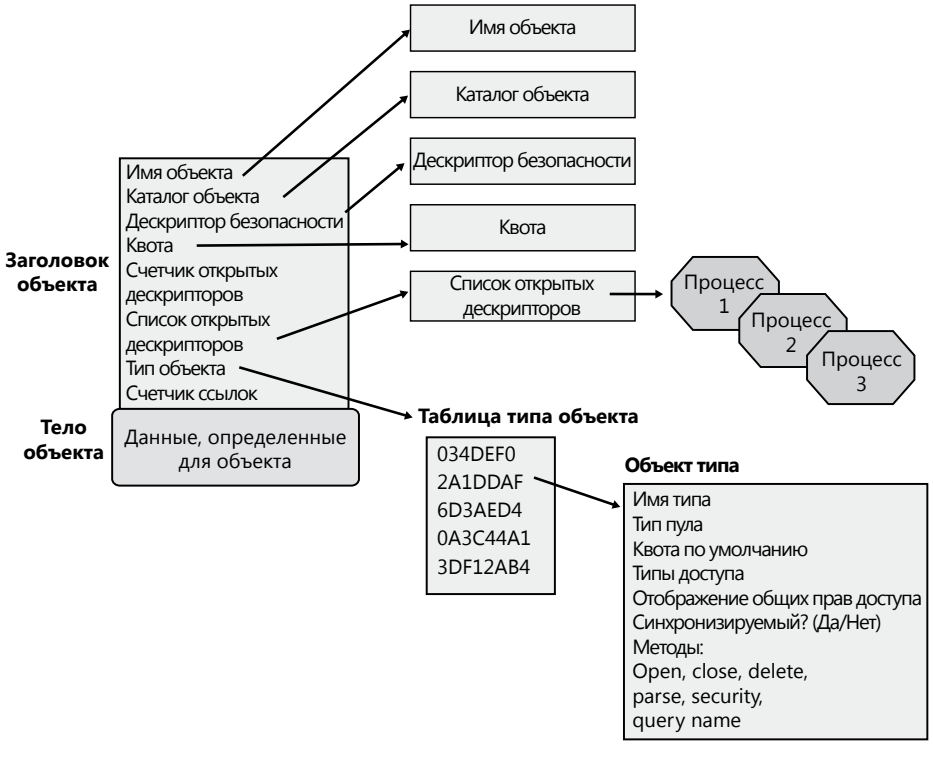


Рис. 3.19. Структура объекта

Заголовки и тела объектов

Диспетчер объектов использует данные, хранящиеся в заголовке объекта для управления объектами независимо от их типа. В табл. 3.9 дано краткое описание полей заголовка объекта, а в табл. 3.10 дано описание полей, которые могут быть в дополнительных подзаголовках объекта.

Кроме заголовка объекта, содержащего информацию, относящуюся к любым объектам, в подзаголовках содержится дополнительная информация, относящаяся к специфическим аспектам объекта. Учтите, что эти структуры находятся по переменному смещению от заголовка объекта, значение которого зависит от количества подзаголовков, связанных с основным заголовком объекта. (За исключением, как ранее указывалось, информации о создателе.) Для каждого имеющегося подзаголовка с целью отображения его существования обновляется поле InfoMask. Когда диспетчер объектов проводит проверку заданного ползаголовка, он проверяет, выставлен ли соответствующий бит в InfoMask, а затем

использует оставшиеся биты для выбора правильного смещения в таблице ObpInfoMaskToOffset, где он находит смещение подзаголовка с начала заголовка объекта.

Эти смещения существуют для всех возможных комбинаций имеющихся в наличии подзаголовков, но, поскольку подзаголовки, если присутствуют, всегда размещаются в фиксированном, постоянном порядке, для заданного заголовка количество возможных мест будет равно максимальному количеству предшествующих ему подзаголовков. Например, поскольку первым всегда следует подзаголовок с информацией об имени, у него есть только одно возможное смещение. С другой стороны, подзаголовок с информацией о дескрипторах (который размещается третьим) имеет три возможных места, поскольку он может размещаться после подзаголовка квоты, а может и не размещаться после этого подзаголовка, который, в свою очередь, мог бы располагаться после информации об имени. Описания всех дополнительных подзаголовков объекта и их мест даны в табл. 3.10. Что касается информации о создателе, то присутствие этого подзаголовка определяется значением флагов заголовка объекта. (Информация об этих флагах дана в табл. 3.12.)

Таблица 3.9. Поля заголовка объекта

Поле	Назначение
Handle count (Счетчик дескрипторов)	Обеспечивает подсчет количества открытых на данный момент дескрипторов к объекту
Pointer count (Счетчик указателей)	Обеспечивает подсчет количества ссылок на объект (включая одну ссылку для каждого дескриптора). Компоненты режима ядра могут ссылаться на объект с помощью указателя без использования дескриптора
Security descriptor (Дескриптор безопасности)	Определяет, кто может воспользоваться объектом и что они могут с ним сделать. Следует учесть, что неименованные объекты не могут обладать безопасностью по определению
Object type index (Индекс типа объекта)	Содержит индекс типа объекта, который содержит атрибуты, общие для объектов данного типа. Таблицей, хранящей все объекты типа, является ObTypeIndexTable
Subheader mask (Маска подзаголовка)	Поразрядная маска, которая описывает, какие из дополнительных структур подзаголовков, рассмотренных в табл. 3.10, присутствуют, за исключением подзаголовка информации о создателе, который, если таковой имеется, всегда предшествует объекту. Поразрядная маска превращается в отрицательное смещение с помощью таблицы ObpInfoMaskToOffset, где каждый подзаголовок связан с однобайтовым индексом, который определяет его позицию относительно других имеющихся подзаголовков
Flags (Флаги)	Характеристики и атрибуты объекта. Полный перечень флагов объекта приведен в табл. 3.12
Lock (Блокировка)	Индивидуальная блокировка объекта, используемая при изменении полей, принадлежащих заголовку этого объекта или любому из его подзаголовков

Таблица 3.10. Дополнительные подзаголовки объекта

Имя	Назначение	Разряд	Местоположение
Creator information (Информация о создателе)	Связывает объект со списком всех объектов одного типа и записывает процесс, который создал проект, наряду с обратной трассировкой	0 (0x1)	Object header-ObpInfoMaskToOffset[0])
Name information (Информация об имени)	Содержит имя объекта, ответственного за обеспечение видимости объекта другим процессам для его совместного использования, и указатель на каталог объекта, предоставляющий иерархическую структуру, в которой хранятся имена объектов	1 (0x2)	Object header - ObpInfoMaskToOffset-ObpInfoMaskToOffset-[InfoMask & 0x3]
Handle information (Информация о дескрипторах)	Содержит базу данных, состоящую из записей (или только из одной записи) для процесса, у которого имеется открытый дескриптор объекта (наряду со счетчиком дескрипторов для каждого процесса)	2 (0x4)	Object header - ObpInfoMaskToOffset-[InfoMask & 0x7]
Quota information (Информация о квоте)	Регистрирует расход ресурсов, забираемых у процесса, когда тот открывает дескриптор объекта	3 (0x8)	Object header - ObpInfoMaskToOffset-[InfoMask & 0xF]
Process information (Информация о процессе)	Содержит указатель на владельца процесса, если объект относится к эксклюзивному. Далее в этой главе будет дана дополнительная информация об эксклюзивных объектах	4 (0x10)	Object header - ObpInfoMaskToOffset-[InfoMask & 0x1F]

Каждый из этих подзаголовков является дополнительным и присутствует только при определенных обстоятельствах, либо в течение запуска системы, либо во время создания объекта. В табл. 3.11 дается описание этих обстоятельств.

И наконец, ряд атрибутов и (или) флагов определяют поведение объекта во время создания или во время определенных операций. Диспетчер объектов получает эти флаги при каждом создании нового объекта в структуре под названием атрибуты объекта. Эта структура определяет имя объекта, корневой каталог объекта, в который он должен быть вставлен, дескриптор безопасности объекта и флаги атрибутов объекта. В табл. 3.12 перечислены различные флаги, которые могут быть связаны с объектом.

Таблица 3.11. Условия, требуемые для присутствия подзаголовков объектов

Имя	Условие
Name information (Информация об имени)	Объект должен быть создан с именем
Quota information (Информация о квоте)	Объект не должен быть создан исходным системным процессом (или процессом простоя)
Process information (Информация о процессе)	Объект должен быть создан с эксклюзивным (exclusive) флагом объекта. (Информация о флагах объекта дана в табл. 3.12.)
Handle information (Информация о дескрипторах)	У типа объекта должен быть установлен флаг поддержки счетчика дескрипторов (handle count). Для файловых объектов, ALPC-объектов, объектов WindowStation и Desktop-объектов этот флаг установлен в их структуре типа объекта
Creator information (Информация о создателе)	У объекта должен быть установлен флаг поддержки списка типа (type list). У объектов драйверов (Driver) этот флаг установлен, если включена проверка Driver Verifier. Но установка глобального флага поддержки списка типа (рассмотренного ранее) создаст это условие для всех объектов, а для Type-объектов этот флаг установлен всегда

ПРИМЕЧАНИЕ

Когда объект создается с помощью API-функций в подсистеме Windows (таких как CreateEvent или CreateFile), вызывающая программа не определяет каких-либо атрибутов объекта — DLL-библиотека подсистемы делает все закулисно. Поэтому все именованные объекты, созданные с помощью Win32, помещаются в каталог BaseNamedObjects, либо в его глобальный экземпляр, либо в экземпляр, созданный для конкретного сеанса, поскольку он является корневым каталогом объектов, определяемых Kernelbase.dll в качестве части структуры атрибутов объекта. Дополнительную информацию о BaseNamedObjects и о его связи с пространством имен, создаваемым для каждого сеанса, можно будет найти далее в этой главе.

Таблица 3.12. Флаги объектов

Флаг атрибутов	Флаг заголовка	Назначение
OBJ_INHERIT	Хранится в записи таблицы дескрипторов	Определяет, будет ли дескриптор объекта унаследован дочерними процессами, и сможет ли процесс воспользоваться функцией DuplicateHandle для создания копии
OBJ_PERMANENT	OB_FLAG_PERMANENT_OBJECT	Определяет способность объекта сохранять поведение, зависящее от рассматриваемых далее подсчетов ссылок
OBJ_EXCLUSIVE	OB_FLAG_EXCLUSIVE_OBJECT	Указывает на то, что объект может использоваться только тем процессом, который его создал

Флаг атрибутов	Флаг заголовка	Назначение
OBJ_CASE_ INSENSITIVE	Хранится в записи таблицы дескрипторов	Указывает на то, что поиск этого объекта в пространстве имен должен вестись без учета регистра символов. Может быть отменен установкой флага *case insensitive* в типе объекта
OBJ_OPENIF	Не сохраняется, используется во время выполнения	Указывает на то, что операция создания объекта с таким же именем должна привести к открытию уже существующего объекта, а не к выдаче ошибки
OBJ_OPENLINK	Не сохраняется, используется во время выполнения	Указывает на то, что диспетчер объектов должен открыть дескриптор символической ссылки, а не целевого объекта
OBJ_KERNEL_ HANDLE	OB_FLAG_KERNEL_ OBJECT	Указывает на то, что дескриптор объекта должен быть *дескриптором ядра* (подробности будут даны чуть позже)
OBJ_FORCE_ ACCESS_ CHECK	Не сохраняется, используется во время выполнения	Указывает на то, что даже при открытии объекта из режима ядра требуется полная проверка прав доступа
OBJ_KERNEL_ EXCLUSIVE	OB_FLAG_KERNEL_ ONLY_ ACCESS	Не дает процессу пользовательского режима открыть дескриптор объекта; используется для защиты раздела объектов /Device/ PhysicalMemory
Отсутствует	OF_FLAG_DEFAULT_ SECURITY_QUOTA	Указывает на то, что дескриптор безопасности объекта использует исходную квоту в 2 Кбайт
Отсутствует	OB_FLAG_SINGLE_ HANDLE_ ENTRY	Указывает на то, что подзаголовок информации о дескрипторах содержит только одну запись и не является базой данных
Отсутствует	OB_FLAG_NEW_ OBJECT	Указывает на то, что объект был создан, но еще не вставлен в пространство имен объектов
Отсутствует	OB_FLAG_DELETED_ INLINE	Указывает на то, что объект удаляется через отложенное удаление рабочего потока

Кроме заголовка у каждого объекта есть его тело, чей формат и содержимое уникальны для типа этого объекта; все объекты одного и того же типа используют один и тот же формат тела. Создавая тип объекта и предоставляя для него соответствующие службы, компонент исполняющей системы может контролировать работу с данными в телах всех объектов данного типа. Поскольку заголовок

объекта имеет статический, заранее известный размер, диспетчер объектов может легко найти заголовок объекта путем простого вычитания размера заголовка из указателя на объект. Как уже ранее указывалось, для получения доступа к подзаголовку диспетчер объектов вычитает еще одно известное значение из указателя на заголовок объекта.

Благодаря нормированным структурам заголовка объекта и подзаголовков диспетчер объектов может предоставить небольшой набор служб, способный работать с атрибутами, сохраненными в любом заголовке объекта, и применимый к объектам любого типа (хотя некоторые общие службы не имеют для определенных объектов никакого смысла). Эти общие службы, часть из которых подсистема Windows делает доступной Windows-приложениям, перечислены в табл. 3.13.

Хотя эти общие службы объектов поддерживаются для объектов всех типов, у каждого объекта есть свои службы создания (create), открытия (open) и запроса (query). Например, система ввода-вывода реализует службу создания файла для своих файловых объектов, а диспетчер процессов реализует службу создания процесса для своих объектов процесса.

Хотя может быть реализована всего одна служба создания объекта, подобная процедура была бы слишком сложной, поскольку набор параметров, необходимых для инициализации, к примеру, файлового объекта, существенно бы отличался от того набора, который понадобился бы для инициализации объекта типа «процесс». Кроме того, диспетчер объектов нес бы дополнительную нагрузку при каждом вызове потоком службы объекта на определение типа объекта, на который ссылается дескриптор, и на вызов соответствующей версии службы.

Таблица 3.13. Общие службы объекта

Служба	Назначение
Close (Закрытия)	Закрывает дескриптор объекта
Duplicate (Дублирования)	Обеспечивает совместное использование объекта путем создания дубликата его дескриптора и предоставления его другому процессу
Make permanent/temporary (Превращения в постоянный или временный)	Изменяет сохранность объекта (рассматриваемую далее)
Query object (Запроса объекта)	Получает информацию о стандартных атрибутах объекта
Query security (Запроса безопасности)	Получает дескриптор безопасности объекта
Set security (Установки безопасности)	Изменяет степень защиты объекта
Wait for a single object (Ожидания одиночного объекта)	Синхронизирует выполнение потока с одним объектом
Signal an object and wait for another (Сообщения об объекте и ожидания другого объекта)	Сообщает об объекте (таком, как событие) и синхронизирует выполнение потока с другим объектом
Wait for multiple objects (Ожидания нескольких объектов)	Синхронизирует выполнение потока с несколькими объектами

Объекты типа

Заголовки объектов содержат данные, общие для всех объектов, которые могут принимать различные значения для каждого экземпляра объекта. Например, у каждого объекта есть уникальное имя и может быть уникальный дескриптор безопасности. Но объекты также содержат некоторые данные, которые остаются постоянными для всех объектов конкретного типа. Например, при открытии дескриптора объекта определенного типа можно выбрать права доступа из набора, присущего этому типу объектов. Исполняющая система предоставляет для объектов типа поток кроме всех прочих доступ к завершению и приостановке, а для объектов типа файл доступ для чтения, записи, добавления и удаления. Еще одним примером атрибута, присущего определенному типу объектов, является синхронизация, которая вскоре будет рассмотрена.

Для экономии памяти диспетчер объектов сохраняет эти статические атрибуты, присущие объектам определенного типа, единожды при создании нового объекта типа. Для записи этих данных он использует свой собственный объект, так называемый объект типа. Как показано на рис. 3.20, если установлен флаг отладки для отслеживания объектов (рассматриваемый далее в разделе «Глобальные флаги Windows»), объект типа также связывает вместе все объекты одного и того же типа (в данном случае типа процесс), позволяя диспетчеру объектов находить эти объекты и вести их подсчет, если это необходимо. Эта функция использует возможность ранее рассмотренного подзаголовка с информацией о создателе.

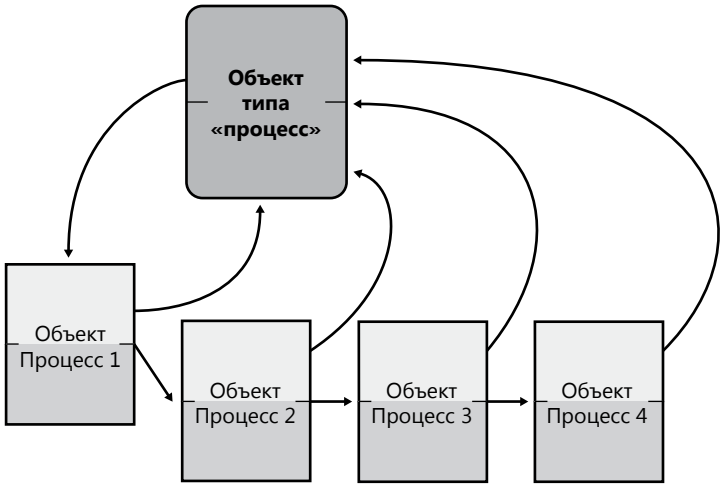


Рис. 3.20. Объекты процессов и объект типа «процесс»

Из пользовательского режима работать с объектами типа нельзя, поскольку диспетчер объектов не предоставляет для них никаких служб. Тем не менее некоторые, определяемые ими атрибуты, видимы с помощью некоторых собственных служб операционной системы и процедур Windows API. Информация, хранящаяся в инициализаторах типа, описана в табл. 3.14.

Таблица 3.14. Поля инициализаторов типа

Атрибут	Назначение
Type name (Имя типа)	Имя объектов данного типа («процесс», «событие», «порт» и т. д.)
Pool type (Тип пула)	Показывает, может ли объектам данного типа выделяться выгружаемая или невыгружаемая память
Default quota charges (Квота по умолчанию)	Значения пулов выгружаемой и невыгружаемой памяти, составляющие по умолчанию квоту процесса.
Valid access mask (Действующая маска доступа)	Виды доступа, которые поток может запросить при открытии дескриптора объекта данного типа («чтение», «запись», «завершение», «приостановка» и т. д.)
Generic access rights mapping (Отображение общих прав доступа)	Отображение четырех общих прав доступа (для чтения, записи, выполнения и всех прав) на права доступа, присущие конкретному типу
Flags (Флаги)	Указывают на то, что объекты не должны иметь имен (например, в случае с объектами типа «процесс»), что в их именах учитывается регистр символов, что они требуют наличие дескриптора безопасности, что они поддерживают обратные вызовы, фильтруемые объектами, и должна ли поддерживаться база данных дескрипторов (подзаголовок информации о дескрипторах) и (или) взаимозависимость списка типов (подзаголовок информации о создателе). Флаг use default object также определяет поведение показанного далее в этой таблице поля default object
Object type code (Код объекта типа)	Используется для описания того, что из себя представляет тип объекта (в отличие от сравнения с известным значением имени). Для файловых объектов значение этого поля устанавливается в 1, для объектов синхронизации — в 2 и для объектов потоков — в 4. Это поле также используется ALPC для хранения информации об атрибуте дескриптора, связанной с сообщением
Invalid attributes (Недопустимые атрибуты)	Задает флаги атрибутов объекта (показанные ранее в табл. 3.12), недопустимые для этого типа объекта
Default object (Объект по умолчанию)	Определяет внутреннее событие диспетчера объектов, которое должно использоваться при ожидании данного объекта, если этого требует создатель объекта типа. Следует учесть, что такие объекты, как File и ALPC-порт, уже содержат свой встроенный диспетчер объектов; в таком случае это поле является смещением в теле объекта. Например, событие внутри структуры FILE_OBJECT встроено в поле под названием Event
Methods (Методы)	Одна или несколько процедур, вызываемых диспетчером объектов автоматически в определенные моменты жизни объекта

ЭКСПЕРИМЕНТ: ПРОСМОТР ЗАГОЛОВКОВ ОБЪЕКТОВ И ОБЪЕКТОВ ТИПА

Структуру данных объекта типа «процесс» можно увидеть в отладчике ядра, предварительно идентифицировав этот объект с помощью команды !process:

```
lkd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS fffffa800279cae0
    SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 00187000  ObjectTable: fffff8a000001920  HandleCount: 541.
    Image: System
```

Выполните команду !object с адресом объекта «процесс» в качестве аргумента:

```
lkd> !object fffffa800279cae0
Object: fffffa800279cae0  Type: (fffffa8002755b60) Process
    ObjectHeader: fffffa800279cab0 (new version)
    HandleCount: 3  PointerCount: 172 3172
```

Учтите, что на 32-разрядной версии Windows заголовок объекта начинается с 0x18 (24 в десятичном формате) байт, предшествующих телу объекта, а на 64-разрядной версии Windows он начинается с 0x30 (48 в десятичном формате) байт, предшествующих телу, то есть с размера самого заголовка объекта. Просмотреть заголовок объекта можно с помощью следующей команды:

```
lkd> dt nt!_OBJECT_HEADER fffffa800279cab0
   +0x000 PointerCount      : 172
   +0x008 HandleCount       : 33
   +0x008 NextToFree        : 0x000000000x00000000'00000003
   +0x010 Lock              : _EX_PUSH_LOCK
   +0x018 TypeIndex         : 0x7 ''
   +0x019 TraceFlags        : 0 ''
   +0x01a InfoMask          : 0 ''
   +0x01b Flags             : 0x2 ''
   +0x020 ObjectCreateInfo  : 0xfffff800'01c53a80 _OBJECT_CREATE_INFORMATION
   +0x020 QuotaBlockCharged : 0xfffff800'01c53a80
   +0x028 SecurityDescriptor: 0xfffff8a0'00004b29
   +0x030 Body              : _QUAD
```

Теперь посмотрим на структуру данных объекта типа, получив его адрес из таблицы ObTypeIndexTable, указанный в записи, связанной с полем TypeIndex структуры данных заголовка объекта:

```
lkd> ?? ((nt!_OBJECT_TYPE**)@@(nt!ObTypeIndexTable))[((nt!_OBJECT_
HEADER*)0xfffffa800279cab0)->TypeIndex]
struct _OBJECT_TYPE * 0xfffffa80'02755b60
   +0x000 TypeList         : _LIST_ENTRY [ 0xfffffa80'02755b60 - 0xfffffa80'02755b60
```

продолжение ⇩

```
]
   +0x010 Name                     : _UNICODE_STRING "Process"
   +0x020 DefaultObject            : (null)
   +0x028 Index                    : 0x70x7 ''
   +0x02c TotalNumberOfObjects     : 0x380x38
   +0x030 TotalNumberOfHandles     : 0x1320x132
   +0x034 HighWaterNumberOfObjects : 0x3d
   +0x038 HighWaterNumberOfHandles : 0x13c
   +0x040 TypeInfo       : _OBJECT_TYPE_INITIALIZER
   +0x0b0 TypeLock       : _EX_PUSH_LOCK
   +0x0b8 Key            : 0x636f7250
   +0x0c0 CallbackList   : _LIST_ENTRY [ 0xfffffa80'02755c20 - 0xfffffa80'02755c20 ]
```

В выведенной информации показано, что структура объекта типа включает имя объекта типа, в ней отслеживается общее количество активных объектов этого типа и пиковое количество дескрипторов и объектов данного типа. В поле CallbackList также отслеживается фильтрация обратных вызовов диспетчера объектов, связанная с этим объектом типа. В поле TypeInfo хранится указатель на структуру данных, в которой хранятся общие для всех объектов этого типа атрибуты, а также указатели на методы объекта типа:

```
lkd> ?? ((nt!_OBJECT_TYPE*)0xfffffa8002755b60)->TypeInfo*)0xfffffa8002755b60)->TypeInfo
   +0x000 Length              : 0x70
   +0x002 ObjectTypeFlags     : 0x4a 'J'
   +0x002 CaseInsensitive     : 0y0
   +0x002 UnnamedObjectsOnly  : 0y1
   +0x002 UseDefaultObject    : 0y0
   +0x002 SecurityRequired    : 0y1
   +0x002 MaintainHandleCount : 0y0
   +0x002 MaintainTypeList    : 0y0
   +0x002 SupportsObjectCallbacks : 0y1
   +0x004 ObjectTypeCode      : 0
   +0x008 InvalidAttributes   : 0xb0
   +0x00c GenericMapping      : _GENERIC_MAPPING
   +0x01c ValidAccessMask     : 0x1fffff
   +0x020 RetainAccess        : 0x101000
   +0x024 PoolType            : 0 ( NonPagedPool )
   +0x028 DefaultPagedPoolCharge    : 0x1000
   +0x02c DefaultNonPagedPoolCharge : 0x528
   +0x030 DumpProcedure    : (null)
   +0x038 OpenProcedure    : 0xfffff800'01d98d58  long  nt!PspProcessOpen+0
   +0x040 CloseProcedure   : 0xfffff800'01d833c4  void  nt!PspProcessClose+0
   +0x048 DeleteProcedure  : 0xfffff800'01d83090  void  nt!PspProcessDelete+0
   +0x050 ParseProcedure   : (null)
   +0x058 SecurityProcedure : 0xfffff800'01d8bb50  long  nt!SeDefaultObjectMethod+0
   +0x060 QueryNameProcedure   : (null)
   +0x068 OkayToCloseProcedure : (null)
```

Синхронизация, являющаяся одним из атрибутов, видимых приложениями Windows, относится к способности потоков синхронизировать их выполнение

путем ожидания перехода объекта из одного состояния в другое. Поток может синхронизироваться с объектами выполняемого задания, процесса, потока, файла, семафора, мьютекса и таймера. Все остальные объекты исполняющей системы синхронизацию не поддерживают. Способность объекта поддерживать синхронизацию основана на трех возможностях:

- Объект исполняющей системы является оболочкой для объекта-диспетчера и содержит заголовок диспетчера, основную структуру, рассматриваемую далее в данной главе, в разделе «Низкоуровневая IRQL-синхронизация».
- Создатель объекта типа требует объект по умолчанию, и диспетчер объектов предоставляет такой объект.
- Объект исполняющей системы имеет встроенный диспетчер объектов, такой как событие, где-нибудь внутри тела объекта, и владелец объекта предоставляет смещение на него диспетчеру объектов при регистрации объекта типа (описано в табл. 3.14).

Методы объекта

Последний атрибут в табл. 3.14, методы, содержит набор внутренних процедур, аналогичных конструкторам и деструкторам C++, то есть процедурам, которые автоматически вызываются при создании или удалении объекта. Диспетчер объектов расширяет этот замысел, вызывая метод объекта и в других ситуациях, например когда кто-нибудь открывает или закрывает дескриптор объекта или когда кто-нибудь пытается изменить защиту объекта. Некоторые типы объектов указывают методы, а некоторые их не указывают, в зависимости от того, как должен использоваться объект того или иного типа.

Когда компонент исполняющей системы создает новый объект типа, он с помощью диспетчера объектов может зарегистрировать один или несколько методов. После этого диспетчер объектов вызывает методы во вполне определенные моменты жизнедеятельности объектов данного типа, обычно при создании объекта, его удалении или каком-нибудь изменении. Методы, поддерживаемые диспетчером объектов, перечислены в табл. 3.15.

Причиной обращения к этим методам объектов служит факт той или иной операции над объектом (закрытия, дублирования, изменения степени безопасности и т. д.). Полное обобщение этих типовых процедур потребовало бы от разработчиков диспетчера объектов предвидения всех типов объектов. Но процедуры для создания объекта типа экспортируются ядром, позволяя внешним компонентам ядра создавать свои собственные типы объектов. Хотя эта функция не фигурирует в документации для разработчиков драйверов, она используется внутри системы компонентом Win32k.sys для определения объектов WindowStation и Desktop. Благодаря возможности расширения методов объектов Win32k.sys определяет свои процедуры для проведения таких операций, как создание и запрос.

Единственным исключением из этого правила является процедура *безопасности*, которая, если не указано иное, выполняет по умолчанию SeDefaultObjectMethod. Этой процедуре не нужно знать внутреннюю структуру объекта,

потому что она имеет дело только с дескриптором безопасности объекта, а вы уже видели, что указатель на дескриптор безопасности хранится в общем заголовке объекта, а не внутри тела объекта. Но если объект требует своей собственной дополнительной проверки безопасности, он может определить свою собственную процедуру безопасности. Еще одной причиной наличия общего метода безопасности является стремление избежать излишней сложности, поскольку многие объекты полагаются при управлении своей безопасностью на монитор безопасности ссылок (security reference monitor).

Таблица 3.15. Методы объекта

Метод	Когда он вызывается
Open (Открыть)	При открытии дескриптора объекта
Close (Закрыть)	При закрытии дескриптора объекта
Delete (Удалить)	Перед удалением объекта диспетчером объектов
Query name (Запросить имя)	Когда поток запрашивает имя объекта, такого как файл, которое существует в пространстве имен производного объекта
Parse (Провести разбор)	При поиске диспетчером объектов имени объекта, которое существует в пространстве имен производного объекта
Dump (Вывести дамп)	Не используется
Okay to close (Подтвердить закрытие)	Когда диспетчер объектов получает указание закрыть дескриптор
Security (Определить степень безопасности)	Когда процесс читает состояние защиты или изменяет защиту такого объекта, как файл, то есть имеет дело со сведениями, существующими в пространстве имен производного объекта

Диспетчер объектов вызывает метод открытия *open* при создании дескриптора объекта, что происходит при создании или открытии объекта. Объекты WindowStation и Desktop предоставляют метод *open*; например, объект типа WindowStation требует такой метод *open*, чтобы Win32k.sys мог совместно использовать часть памяти с процессом, который служит в качестве пула памяти, связанного с рабочим столом.

Пример использования метода закрытия *close* можно найти в системе ввода-вывода. Диспетчер ввода-вывода регистрирует метод *close* для объекта типа «файл», а диспетчер объектов вызывает метод *close* при каждом закрытии дескриптора файлового объекта. Этот метод *close* проверяет, не владеет ли процесс, закрывающий дескриптор файла, какими-либо незавершенными блокировками, касающимися этого файла, и если такие блокировки имеются, он их удаляет. Проверка наличия блокировок, связанных с файлами, не является обязанностью, которую должен или может выполнять сам диспетчер объектов.

Перед удалением временного объекта из памяти диспетчер объектов вызывает метод *delete*, если такой метод был зарегистрирован. Диспетчер памяти, к примеру, регистрирует метод *delete* для объекта типа «раздел», и этот метод освобождает физические страницы, использовавшиеся в разделе. Он также проверяет перед удалением объекта типа «раздел» факт удаления любых внутренних

структур данных, выделенных разделу диспетчером памяти. И, опять же, диспетчер объектов не может справиться с этой работой, поскольку ему ничего не известно о внутренней работе диспетчера памяти. Методы удаления для других типов объектов выполняют аналогичные функции.

Метод разбора *parse* (так же, как и метод запроса имени *query name*) позволяет диспетчеру объектов уступать управление поиском объекта производному диспетчеру объектов, если он найдет объект, существующий за пределами пространства имен диспетчера объектов. Когда диспетчер объектов ищет имя объекта, он приостанавливает свой поиск, когда обнаруживает объект в пути, который он связывает с методом *parse*. Диспетчер объектов вызывает метод *parse*, передавая ему оставшуюся часть от имени объекта, поиском которого он занимается. Кроме пространства имен диспетчера объектов в Windows есть еще два пространства имен: пространство имен реестра, реализуемое диспетчером конфигурации, и пространство имен файловой системы, реализуемое диспетчером ввода-вывода с помощью драйверов файловой системы (см. главу 4).

Например, когда процесс открывает дескриптор объекта \Device\HarddiskVolume1\docs\resume.doc, диспетчер объектов проходит по дереву его имени до тех пор, пока не дойдет до имени объекта устройства HarddiskVolume1. Он видит, что с этим объектом связан метод разбора *parse*, и вызывает этот метод, передавая ему остаток имени объекта, поиск которого он вел, в данном случае это строка docs\resume.doc. Метод *parse* для объектов устройств является процедурой ввода-вывода, поскольку объекты типа «устройство» определялись диспетчером ввода-вывода и им же регистрировался для них метод *parse*. Процедура *parse*, задаваемая диспетчером ввода-вывода, получает строку имени и передает ее соответствующей файловой системе, которая ищет файл на диске и открывает его.

Метод безопасности *security*, который также использует система ввода-вывода, аналогичен методу *parse*. Он вызывается в том случае, если поток пытается запросить или изменить информацию безопасности, защищающую файл. Эта информация для файлов отличается от такой же информации для других объектов, поскольку информация о безопасности хранится в самом файле, а не в памяти. Поэтому для поиска информации о безопасности и ее считывании или изменении должна быть вызвана система ввода-вывода.

И наконец, метод подтверждения закрытия *okay-to-close* используется в качестве дополнительного уровня защиты от злонамеренного или ошибочного закрытия дескрипторов, используемых для системных целей. Например, у каждого процесса есть дескриптор на объект Desktop или объекты, с которыми у его потока или потоков имеются видимые окна.

В соответствии со стандартной моделью безопасности такие потоки могут закрывать свои дескрипторы на их рабочие столы, поскольку у процесса есть полный контроль над его собственными объектами. По этому сценарию потоки остаются без связанного с ними рабочего стола, что является нарушением многооконной модели. Для предотвращения подобного поведения Win32k.sys регистрирует процедуру *okay-to-close* для объектов рабочего стола Desktop и объектов WindowStation.

Дескрипторы объекта и таблица дескрипторов процесса

Когда процесс создает или открывает объект по его имени, он получает дескриптор, дающий ему доступ к объекту. Ссылаться на объект по его дескриптору быстрее, чем использовать его имя, поскольку диспетчер объектов может не заниматься поиском по имени и находить объект напрямую. Процессы могут также получать дескрипторы объектов путем наследования дескрипторов во время создания процесса (когда создатель устанавливает флаг наследования дескрипторов при вызове функции CreateProcess, и дескриптор помечен как наследуемый либо в процессе своего создания, либо после создания путем использования Windows-функции SetHandleInformation) или путем получения продублированного дескриптора от другого процесса (см. Windows-функцию DuplicateHandle).

Все процессы пользовательского режима должны иметь дескриптор объекта, прежде чем их потоки смогут использовать объект. Использование дескрипторов для управления системными ресурсами — идея не новая. Например, библиотеки времени выполнения C и Pascal (более старого языка программирования, аналогичного Delphi) возвращают дескрипторы открытых файлов. Дескрипторы служат в качестве косвенных указателей на ресурсы системы; эта косвенность не дает прикладным программам напрямую манипулировать структурами системных данных.

ПРИМЕЧАНИЕ

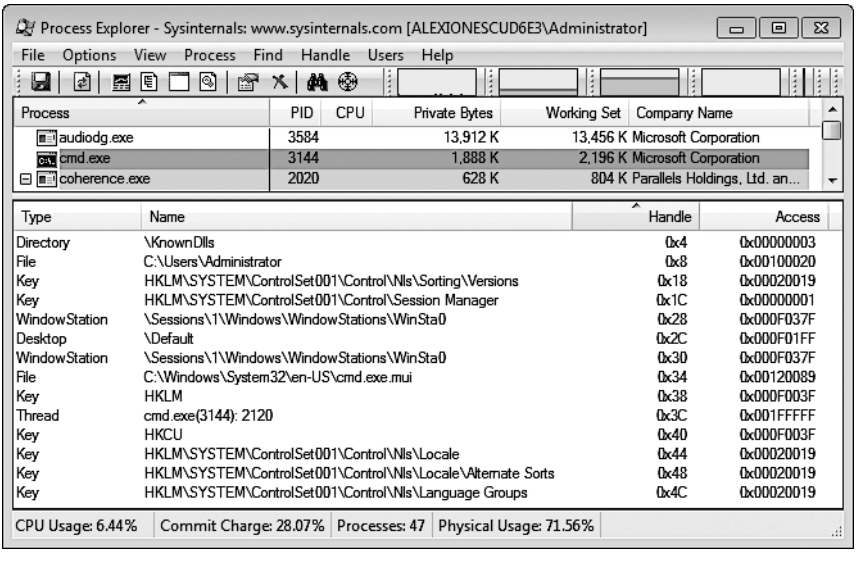
Компоненты исполняющей системы и драйверы устройств могут обращаться к объектам непосредственно, поскольку они запущены в режиме ядра и поэтому имеют доступ к структурам объекта в системной памяти. Но они должны объявить о своем использовании объекта, увеличив значение счетчика ссылок, чтобы объект не мог быть удален из памяти, пока он все еще используется. (Подробности даны далее в разделе «Сохранение объектов».) Однако для успешного использования объекта драйверы устройств должны знать определение внутренней структуры объекта, а для многих объектов она не предоставляется. Взамен драйверам устройств рекомендуется использовать соответствующие API-функции ядра для изменения или чтения информации из объекта. Например, хотя драйверы устройств могут получить указатель на объект типа «процесс» (EPROCESS), его структура им не известна, и должны быть использованы API-функции вида Ps*. Для других объектов непрозрачен сам тип (это касается большинства объектов исполняющей системы, в которые заключен диспетчер объектов, в качестве примеров можно привести события или мьютексы). Для таких объектов драйверы должны использовать такие же системные вызовы, которые в конечном итоге используются приложениями пользовательского режима (такие как ZwCreateEvent), и использовать дескрипторы, а не указатели на объекты.

Дескрипторы объектов дают дополнительные преимущества. Во-первых, за исключением того, к чему они относятся, нет никакой разницы между дескрипторами файла, события и процесса. Эта схожесть обеспечивает единый интерфейс для ссылки на объекты, независимо от их типа. Во-вторых, диспетчер объектов имеет исключительное право на создание дескрипторов и на определение местоположения объекта, который относится к дескриптору. Это означает,

что диспетчер объектов может проверять каждое действие в пользовательском режиме, влияющее на объект, чтобы убедиться в том, что профиль безопасности вызывающей программы разрешает проводить запрашиваемую операцию над данным объектом.

ЭКСПЕРИМЕНТ: ПРОСМОТР ОТКРЫТЫХ ДЕСКРИПТОРОВ

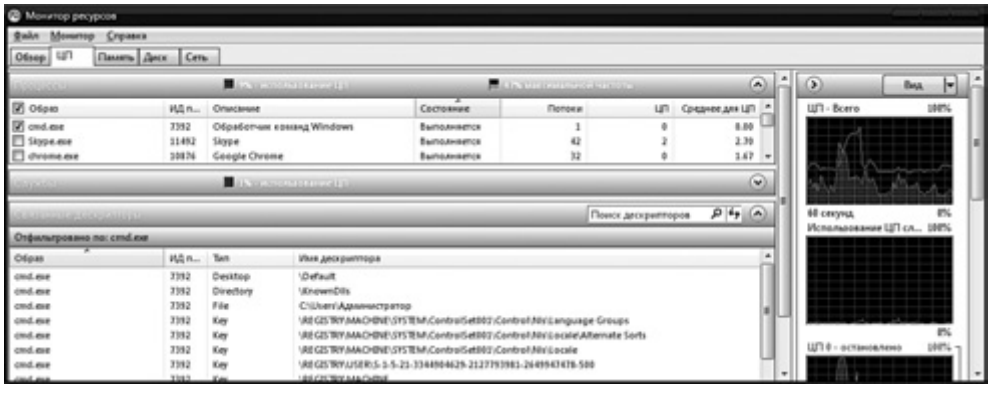
Запустите Process Explorer и убедитесь, что нижняя панель включена и настроена на показ открытых дескрипторов. (View (Вид) ▸ Lower Pane View (Просмотр нижней панели) ▸ Handles (Дескрипторы)). После этого откройте окно командной строки и просмотрите таблицу дескрипторов для нового процесса Cmd.exe. Вы должны увидеть дескриптор открытого файла для текущего каталога. Например, предположим, что текущим является каталог C:\Users\Administrator, тогда Process Explorer покажет следующее.



Теперь поставьте Process Explorer на паузу, нажав клавишу Пробел или щелкнув на пунктах View (Вид) ▸ Update Speed (Изменить скорость) ▸ Pause (Пауза). Затем измените текущий каталог с помощью команды cd и нажмите клавишу F5, чтобы обновить отображаемую информацию. Вы увидите в Process Explorer, что дескриптор предыдущего текущего каталога закрыт и открыт новый дескриптор для нового текущего каталога. Предыдущий дескриптор выделен красным цветом, а новый дескриптор выделен зеленым цветом.

Свойство выделения разным цветом, имеющееся в Process Explorer, делает заметнее изменения в таблице дескрипторов. Например, если процесс допускает утечку дескрипторов, просмотр таблицы дескрипторов с помощью Process Explorer может быстро показать, какой дескриптор или какие дескрипторы были открыты, но не были закрыты. (Обычно виден длинный список дескрипторов для одного и того же объекта.) Эта информация поможет программисту обнаружить утечку дескрипторов.

Монитор ресурсов также показывает открытые (именованные) дескрипторы для процессов, выбранных путем установки флажков напротив их имен. Вот как выглядят дескрипторы открытого окна командной строки.



Таблицу открытых дескрипторов можно также вывести, используя средство командной строки Handle из серии программных продуктов Sysinternals. Посмотрите, к примеру, на следующий, частично показанный вывод, полученный с помощью средства Handle при изучении дескрипторов файловых объектов, находящихся в таблице дескрипторов для процесса Cmd.exe до и после изменения каталога. По умолчанию Handle отфильтровывает нефайловые дескрипторы, пока не будет использован ключ –a, который приводит к выводу всех дескрипторов в процессе, аналогично Process Explorer.

```
C:\>handle -p cmd.exe
Handle v3.46
Copyright (C) 1997-2011 Mark Russinovich
Sysinternals - www.sysinternals.com
------------------------------------------------------------------------------
cmd.exe pid: 5124 Alex-Laptop\Alex Ionescu
   3C: File  (R-D)   C:\Windows\System32\en-US\KernelBase.dll.mui
   44: File  (RW-)   C:\
C:\>cd windows
C:\Windows>handle -p cmd.exe
Handle v3.46
Copyright (C) 1997-2011 Mark Russinovich
Sysinternals - www.sysinternals.com
------------------------------------------------------------------------------
cmd.exe pid: 5124 Alex-Laptop\Alex Ionescu
   3C: File  (R-D)   C:\Windows\System32\en-US\KernelBase.dll.mui
   40: File  (RW-)   C:\Windows
```

Дескриптор объекта является индексом в таблице дескрипторов, относящейся к конкретному процессу. Этот индекс указывается исполнительным блоком процесса (EPROCESS), который рассматривается в главе 5. Первый индекс дескриптора имеет значение 4, второй — 8 и т. д. Таблица дескрипторов процесса содержит указатели на все объекты, которые процесс открыл для своей работы. Таблицы дескрипторов реализованы по древовидной схеме, подобной той, которую реализует блок управления памятью x86 для перевода виртуальных адресов в физические, которая дает максимальное значение, превышающее 16 000 000 дескрипторов на процесс.

ПРИМЕЧАНИЕ

В древовидной схеме таблиц таблица верхнего уровня может содержать страницу, заполненную указателями на таблицы среднего уровня, что позволяет иметь более половины миллиарда дескрипторов. Но чтобы поддержать совместимость со схемой дескрипторов, имевшейся в Windows 2000, и унаследовать ограничение в 16 777 216 дескрипторов, таблица верхнего уровня содержит не более 32 указателей на таблицы среднего уровня, устанавливая для более новых версий Windows тот же предел.

При создании процесса выделяется только таблица дескрипторов самого низкого уровня, другие уровни создаются по мере необходимости. Таблица дескрипторов нижнего уровня состоит из такого количества записей, которое может поместиться на странице минус одна запись, которая используется для контроля дескрипторов. Например, для систем x86 страница составляет 4096 байт и поделена на записи таблицы дескрипторов размером 8 байт, которых получается 512 минус 1, то есть всего 511 записей в таблице дескрипторов самого низкого уровня. Таблица дескрипторов среднего уровня содержит полную страницу указателей на таблицы нижнего уровня, поэтому количество таблиц дескрипторов нижнего уровня зависит от размера страницы и размера указателя для платформы. Схема таблицы дескрипторов в системе Windows показана на рис. 3.21.

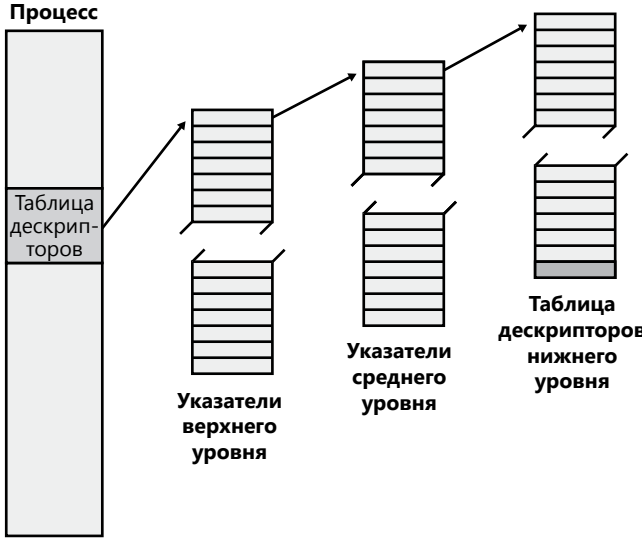


Рис. 3.21. Архитектура таблицы дескрипторов процесса Windows

ЭКСПЕРИМЕНТ: СОЗДАНИЕ МАКСИМАЛЬНОГО КОЛИЧЕСТВА ДЕСКРИПТОРОВ

Тестовая программа Testlimit из коллекции Sysinternals предоставляет возможность открывать дескрипторы объекта до тех пор, пока она не сможет больше открыть ни одного дополнительного дескриптора. Этим можно воспользоваться, чтобы посмотреть, сколько дескрипторов может быть создано

отдельно взятым процессом на вашей системе. Поскольку под таблицы дескрипторов выделяется место в выгружаемом пуле памяти, до достижения максимального количества дескрипторов, доступного для создания в рамках отдельно взятого процесса, вы можете выйти за рамки выгружаемого пула. Чтобы увидеть, сколько дескрипторов можно создать на вашей системе, выполните следующие действия:

1. Загрузите с адреса http://live.sysinternals.com/WindowsInternals исполняемый файл Testlimit, необходимой вам версии для 32- или 64-разрядной версии Windows.
2. Запустите программу Process Explorer, щелкните на пункте View (Вид) ▸ System Information (Информация о системе) ▸ Memory (Память). Обратите внимание на текущий и максимальный размер выгружаемого пула. Чтобы показать максимальный размер пула, Process Explorer должен быть правильно настроен на доступ к символам образа ядра, Ntoskrnl.exe. Оставьте это отображение системной информации работающим, чтобы можно было следить за использованием пула при запуске программы Testlimit.
3. Откройте окно командной строки.
4. Запустите программу Testlimit с ключом –h (путем ввода команды testlimit –h). Когда программа Testlimit не сможет открыть новый дескриптор, она покажет общее количество тех дескрипторов, которые ей удалось создать. Если количество будет меньше, чем примерно 16 миллионов, значит, вы, наверное, вышли за пределы выгружаемого пула памяти еще до достижения теоретического лимита дескрипторов на один процесс.
5. Закройте окно командной строки, благодаря чему будет прекращено выполнение процесса Testlimit и будут закрыты все открытые им дескрипторы. ■

Как показано на рис. 3.22, на системах x86 каждая запись дескриптора состоит из структуры с двумя 32-разрядными элементами: указателем на объект (с флагами) и маской предоставленных прав доступа. На 64-разрядных системах запись таблицы дескрипторов имеет длину 12 байт: 64-разрядный указатель на заголовок объекта и 32-разрядная маска доступа (см. главу 6).

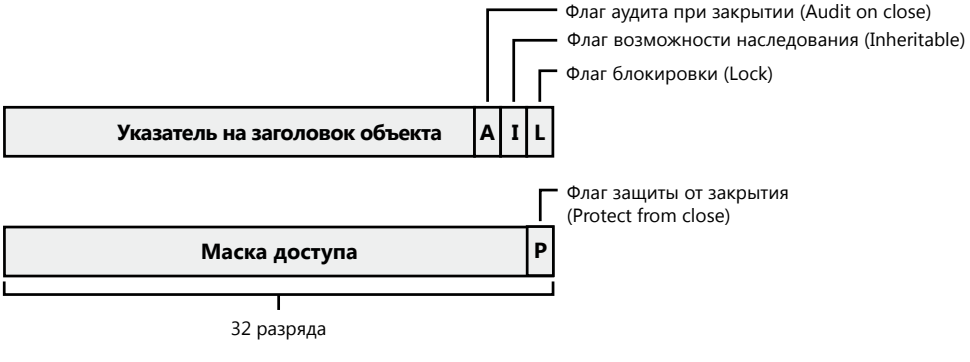


Рис. 3.22. Структура записи в таблице дескрипторов

ЭКСПЕРИМЕНТ: ПРОСМОТР ТАБЛИЦЫ ДЕСКРИПТОРОВ С ПОМОЩЬЮ ОТЛАДЧИКА ЯДРА

В команде !handle отладчика ядра используются три аргумента:

```
!handle <индекс дескриптора> <флаги> <идентификатор процесса>
```

Индекс дескриптора идентифицирует запись дескриптора в таблице дескрипторов. (Нуль означает «показать все дескрипторы».) Индекс первого дексритора имеет значение 4, второго — 8 и т. д. Например, после ввода команды !handle 4 будет показан первый дескриптор для текущего процесса.

Флаги можно указать в виде поразрядной маски, где разряд 0 означает «показать только информацию в записи дескриптора», разряд 1 означает «показать свободные (то есть неиспользуемые) дескрипторы, а разряд 2 означает «показать информацию об объекте, на который ссылается дескриптор». Следующая команда приводит к показу всех подробностей о таблице дескрипторов для процесса с идентификатором 0x62C:

```
lkd> !handle 0 7 62c
processor number 0, process 000000000000062c
Searching for Process with Cid == 62c
PROCESS fffffa80052a7060
    SessionId: 1  Cid: 062c    Peb: 7fffffdb000  ParentCid: 0558
    DirBase: 7e401000  ObjectTable: fffff8a00381fc80  HandleCount: 111.
    Image: windbg.exe
Handle table at fffff8a0038fa000 with 113 Entries in use
0000: free handle, Entry address fffff8a0038fa000, Next Entry 00000000ffffffe
0004: Object: fffff8a005022b70  GrantedAccess: 00000003 Entry: fffff8a0038fa010
Object: fffff8a005022b70  Type: (fffffa8002778f30) Directory
    ObjectHeader: fffff8a005022b40fffff8a005022b40 (new version)
        HandleCount: 25  PointerCount: 63
           Directory Object: fffff8a000004980  Name: KnownDlls
0008: Object: fffffa8005226070  GrantedAccess: 00100020 Entry: fffff8a0038fa020
Object: fffffa8005226070  Type: (fffffa80027b3080) File
   ObjectHeader: fffffa8005226040fffffa8005226040 (new version)
    HandleCount: 1  PointerCount: 1
     Directory Object: 00000000  Name: \Program Files\Debugging Tools for Windows (x64)
{HarddiskVolume2}
```

Первым флагом является бит блокировки, показывающий, что запись в настоящее время используется. Вторым флагом является указатель на возможность наследования, то есть он показывает, получат ли процессы, созданный данным процессом, копию этого дескриптора в свои таблицы дескрипторов. Как уже отмечалось, наследование дескрипторов может быть указано при создании дескриптора или после него с помощью функции SetHandleInformation. Третий флаг показывает, должно ли закрытие объекта генерировать контрольное сообщение (флаг не показывается в Windows, диспетчер объектов использует его для внутренних нужд). Бит защиты от закрытия хранится в неиспользованной части маски доступа и показывает, разрешено ли вызывающей программе закрыть

этот дескриптор (флаг может быть установлен с помощью системного вызова NtSetInformationObject).

Системным компонентам и драйверам устройств зачастую нужно открывать дескрипторы объектов, к которым не должны иметь доступ приложения пользовательского режима. Это делается путем создания дескрипторов в таблице дескрипторов ядра (внутренняя ссылка на которую осуществляется по имени ObpKernelHandleTable). Дескрипторы в этой таблице доступны только из режима ядра и в контексте любого процесса. Это означает, что функция режима ядра может сослаться на дескриптор в контексте любого процесса, не оказывая отрицательного влияния на производительность системы. Диспетчер объектов распознает ссылки на дескрипторы из таблицы дескрипторов ядра, когда установлен старший бит дескриптора, то есть когда ссылки на дескрипторы из таблицы дескрипторов ядра имеют значение, больше чем 0x80000000. Таблица дескрипторов ядра также служит в качестве таблицы дескрипторов для процесса System, и все дескрипторы, созданные процессом System (например, кодом, запущенным в системных потоках), автоматически помечаются как дескрипторы ядра, поскольку они размещаются в таблице дескрипторов ядра по определению.

ЭКСПЕРИМЕНТ: ПОИСК ОТКРЫТЫХ ФАЙЛОВ С ПОМОЩЬЮ ОТЛАДЧИКА ЯДРА

Хотя для поиска дескрипторов открытых файлов можно воспользоваться такими средствами, как Process Explorer, Handle и OpenFiles.exe, они недоступны при просмотре аварийного дампа или удаленном анализе системы. Вместо них для поиска дескрипторов, отрытых для файлов на том или ином томе, можно воспользоваться командой !devhandles.

1. Сначала нужно выбрать букву диска, представляющего интерес, и получить указатель на его объект Device. Для этого, как показано ниже, можно воспользоваться командой !object:

```
1: kd> !object \Global??\C:
Object: fffff8a00016ea40  Type: (fffffa8000c38bb0) SymbolicLink
    ObjectHeader: fffff8a00016ea10 (new version)
    HandleCount: 0  PointerCount: 1
    Directory Object: fffff8a000008060  Name: C:
    Target String is '\Device\HarddiskVolume1'
    Drive Letter Index is 3 (C:)
```

2. Затем нужно воспользоваться командой !object, чтобы получить объект Device для нужного имени тома:

```
1: kd> !object \Device\HarddiskVolume1
Object: fffffa8001bd3cd0  Type: (fffffa8000ca0750) Device
```

3. Теперь можно воспользоваться указателем на объект Device, вставив его в команду !devhandles. Каждый показанный объект указывает на файл:

```
!devhandles fffffa8001bd3cd0
Checking handle table for process 0xfffffa8000c819e0
Kernel handle table at fffff8a000001830 with 434 entries in use

PROCESS fffffa8000c819e0
```

```
    SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 00187000  ObjectTable: fffff8a000001830  HandleCount: 434.
    Image: System
0048: Object: fffffa8001d4f2a0  GrantedAccess: 0013008b Entry: fffff8a000003120
Object: fffffa8001d4f2a0  Type: (fffffa8000ca0360) File
    ObjectHeader: fffffa8001d4f270 (new version)
        HandleCount: 1  PointerCount: 19
        Directory Object: 00000000  Name: \Windows\System32\LogFiles\WMI\
RtBackup\EtwRTEventLog-Application.etl {HarddiskVolume1}
```

Резервные объекты

Поскольку с помощью объектов представлено все, от событий до файлов и до сообщений, которыми процессы обмениваются между собой, способность приложений и кода ядра создавать объекты является основой для нормального и желаемого поведения в процессе выполнения любого фрагмента кода Windows. Если назначить объект не удается, это обычно служит причиной чего угодно, от утраты функционирования (процесс не может открыть файл) до утраты данных или попадания в аварийное состояние (процесс не может назначить объект синхронизации). Хуже того, в определенных ситуациях создание отчетов об ошибках, связанных с отказами в создании объектов, может само требовать назначения новых объектов. Для разрешения подобных ситуаций в Windows реализуются два специальных резервных объекта: резервный объект асинхронного вызова процедуры в пользовательском режиме — User APC reserve object и резервный объект пакета завершения ввода-вывода — I/O Completion packet reserve object. Следует заметить, что сам механизм резервных объектов полностью открыт для расширения и в будущих версиях Windows могут быть добавлены и другие типы резервных объектов. По большому счету, резервный объект является механизмом, позволяющим любой структуре данных режима ядра быть заключенной в оболочку объекта (с соответствующим дескриптором, именем и мерами обеспечения безопасности) для дальнейшего использования.

Как уже говорилось ранее в разделе, посвященном APC-вызовам, эти вызовы используются для таких операций, как приостановка, завершение работы и завершение ввода-вывода, а также для связи между приложениями пользовательского режима, которым нужно обеспечивать асинхронные обратные вызовы. Когда приложение пользовательского режима запрашивает User APC, нацеленный на другой поток, оно использует API-функцию QueueUserApc из библиотеки Kernelbase.dll, которая осуществляет системный вызов NtQueueUserApcThread. В ядре этот системный вызов пытается выделить часть выгружаемого пула для хранения структуры управляющего объекта KAPC, связанной с APC-вызовом. Когда испытывается дефицит памяти, эта операция терпит неудачу, мешая доставке APC, что, в зависимости от того, какой APC-вызов для какой цели использовался, может привести к потере данных или к нарушению функционирования системы.

Чтобы предотвратить такую возможность, приложение пользовательского режима может в самом начале своей работы воспользоваться системным вызовом NtAllocateReserveObject, чтобы запросить у ядра заблаговременного выделения

памяти под KAPC-структуру. Затем приложение использует другой системный вызов, NtQueueUserApcThreadEx, содержащий дополнительный параметр, используемый для хранения дескриптора резервного объекта. Вместо размещения новой структуры, ядро пытается получить резервный объект (путем установки его бита использования InUse в true) и воспользоваться им, пока у KAPC-объекта не минует в нем надобность, в случае чего резервный объект будет возвращен системе. В данное время, в целях недопущения сторонними разработчиками неправильного управления системными ресурсами, API-функции, работающие с резервными объектами, доступны только внутри системы, через системные вызовы компонентов операционной системы. Например, RPC-библиотека использует резервные APC-объекты, чтобы гарантировать, что асинхронные обратные вызовы все же получат возможность возвращения в случае дефицита памяти.

Аналогичный сценарий может сложиться, когда приложению нужна гарантированная доставка сообщения или пакета порта завершения ввода-вывода. Обычно пакеты отправляются с помощью API-функции PostQueuedCompletionStatus из библиотеки Kernelbase.dll, которая вызывает API-функцию NtSetIoCompletion. Так же, как и в случае с пользовательскими APC-вызовами, ядро должно разместить структуру диспетчера ввода-вывода, чтобы в ней содержалась информация пакета завершения, и если размещение потерпит неудачу, пакет не сможет быть создан. При наличии резервных объектов приложение может воспользоваться в начале своей работы API-функцией NtAllocateReserveObject, чтобы получить заранее размещенный в ядре пакет завершения ввода-вывода, а для предоставления дескриптора этому резервному объекту, гарантирующего успешное прохождение, может быть использован системный вызов NtSetIoCompletionEx. Точно так же, как и в случае с резервными объектами User APC, эта функциональная возможность зарезервирована для системных компонентов и используется как RPC-библиотекой, так и Windows-службой Peer-To-Peer BranchCache (дополнительные сведения о работе с сетями даны в главе 7 «Сеть») для гарантированного завершения асинхронных операций ввода-вывода.

Безопасность объекта

При открытии файла нужно указать, что с ним собираются делать, читать или записывать данные. При попытке записи в файл, открытый для доступа только по чтению, будет получена ошибка. Точно так же в исполняющей системе, когда процесс создает объект или открывает дескриптор уже существующего объекта, процесс должен указать набор *желаемых прав доступа*, то есть что он хочет делать с этим объектом. Он может запросить либо набор стандартных прав доступа (например, по чтению, записи и выполнению), применимый ко всем объектам типа, либо конкретные права доступа, изменяющиеся в зависимости от типа объекта. Например, процесс может запросить доступ к файловому объекту для удаления или для добавления. Аналогично этому, он может потребовать возможность приостановки или завершения работы объекта потока.

Когда процесс открывает дескриптор объекта, диспетчер объектов вызывает *монитор безопасности ссылок* (security reference monitor), ту часть системы безопасности, которая работает в режиме ядра, передавая ему от процесса набор желаемых прав доступа. Монитор безопасности ссылок проверяет, разрешает ли дескриптор безопасности объекта тот вид доступа, который запрашивается процес-

сом. Если да, монитор возвращает набор *выделенных прав доступа*, которым разрешено пользоваться процессу, а диспетчер объектов сохраняет права доступа в создаваемом им дескрипторе объекта. Вопрос о том, как именно система безопасности определяет, кто получает права и к каким объектам, рассматривается в главе 6.

Впоследствии, когда потоки процесса используют дескриптор через системный вызов, диспетчер объектов может быстро проверить, сохранен ли в дескрипторе объекта набор выделенных прав доступа, соответствующий использованию применительно к той службе объекта, которая была вызвана потоком. Например, если вызывающая программа запрашивает доступ по чтению к объекту раздела, но затем вызывает службу записи в этот объект, служба ответит отказом.

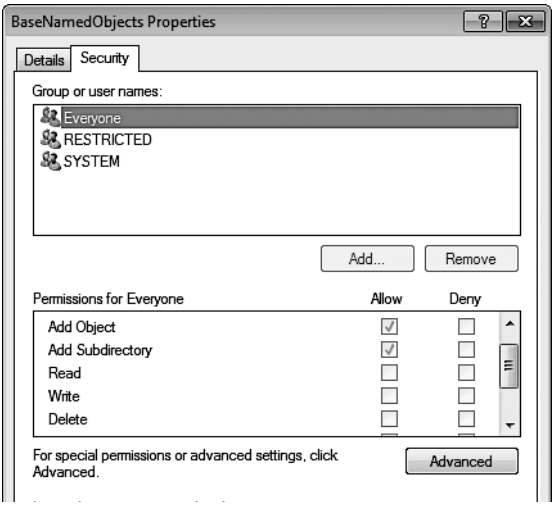
Windows также поддерживает расширенные Ex-версии (от слова Extended) API-функций — CreateEventEx, CreateMutexEx, CreateSemaphoreEx, — которые добавляют еще один аргумент для указания маски доступа. Это дает возможность приложениям правильно использовать списки управления избирательным доступом (discretionary access control lists, DACL) для защиты их объектов, не разрушая при этом их способности использовать API-функции создания объектов для открытия их дескрипторов. А почему бы клиентскому приложению просто не воспользоваться функцией OpenEvent, поддерживающей желаемый аргумент доступа? Использование API-функций открытия объекта приводит к присущим этим функциям соревновательным состояниям, когда речь идет об отказе в вызове открытия, то есть, когда клиентское приложение попыталось открыть событие еще до того, как оно было создано. Во многих приложениях такого типа в случае отказа API-функция открытия следует за API-функцией создания. К сожалению, гарантированного способа сделать эту операцию создания атомарной, иными словами, происходящей только один раз, не существует. В действительности, API-функции создания могут выполняться сразу несколькими потоками и (или) процессами параллельно, и все они могут пытаться создать событие в одно и то же время. Эти соревновательные условия и дополнительные сложности, требуемые для того, чтобы попытаться справиться с проблемой, делают использование API-функций открытия объекта неприемлемым решением проблемы, именно поэтому вместо них должны использоваться API-функции Ex.

ЭКСПЕРИМЕНТ: ПРОСМОТР БЕЗОПАСНОСТИ ОБЪЕКТА

Различные права доступа к объекту можно посмотреть либо с помощью Process Explorer, WinObj, либо с помощью AccessCheck. Все эти средства входят в набор, предоставляемый Sysinternals. Давайте рассмотрим различные способы вывода списка контроля доступа объекта (access control list, ACL).

Для перехода к любому объекту системы, включая каталоги объектов, можно воспользоваться средством WinObj, щелкнув правой кнопкой мыши на объекте и выбрав пункт Properties (Свойства). Например, выберите каталог BaseNamedObjects, выберите пункт Properties (Свойства) и щелкните на вкладке Security (Безопасность). Появится диалоговое окно, похожее на то, что показано далее.

Изучая настройки в диалоговом окне, можно увидеть, например, что группа Everyone (Все) не имеет права на удаление каталога, а вот группа SYSTEM такое право имеет (поскольку там будут хранить свои объекты службы сеанса 0 с правами SYSTEM).



Вместо использования WinObj можно просмотреть таблицу дескрипторов процесса, используя средство Process Explorer, как показано в рассмотренном ранее эксперименте «Просмотр открытых дескрипторов» (с. 193). Посмотрите на таблицу дескрипторов для процесса Explorer.exe. Вы должны заметить в каталоге \Sessions\n\BaseNamedObjects дескриптор объекта Directory. Можно дважды щелкнуть на дескрипторе объекта, а затем щелкнуть на вкладке Security (Безопасность) и увидеть аналогичное диалоговое окно (в котором будет больше пользователей и выделенных прав). Process Explorer не может декодировать права доступа конкретного объекта каталога, поэтому вы увидите только общие права доступа.

И наконец, для запроса информации о безопасности любого объекта можно воспользоваться средством AccessCheck с использованием ключа –o, как показано в следующем выводе. Учтите, что использование AccessCheck покажет вам также уровень целостности объекта (см. главу 6).

```
C:\Windows>accesschk -o \Sessions\1\BaseNamedObjects
Accesschk v5.02 - Reports effective permissions for securable objects
Copyright (C) 2006-2011 Mark Russinovich
Sysinternals - www.sysinternals.com
\sessions\2\BaseNamedObjects
  Type: Directory
  RW NT AUTHORITY\SYSTEM
  RW NTDEV\markruss
  RW NTDEV\S-1-5-5-0-5491067-markruss
  RW BUILTIN\Administrators
  R  Everyone
        NT AUTHORITY\RESTRICTED
```

Сохранение объектов

Есть два вида объектов: временные и постоянные. Большинство объектов являются временными, то есть они живут, пока используются, и освобождаются по миновании надобности. Постоянные объекты живут до тех пор, пока не будут

освобождены явным образом. Поскольку большинство объектов являются временными, в остальной части этого раздела описывается, как диспетчер объектов реализует сохранность объекта, то есть сохраняет временные объекты только на время их использования, а затем их удаляет. Поскольку все процессы пользовательского режима, имеющие доступ к объекту, должны сначала открыть дескриптор объекта, диспетчер объектов может легко отследить, сколько имеется таких процессов, и выявить даже те из них, которые используют объект. Отслеживание этих дескрипторов представляет одну часть реализации сохранения объектов. Диспетчер объектов реализует сохранение объекта в два этапа. Первый этап называется сохранением имени, и он управляется с помощью количества открытых дескрипторов существующего объекта. При каждом открытии процессом дескриптора объекта диспетчер объектов увеличивает показатель счетчика открытых дескрипторов в заголовке объекта. Как только процессы завершают использование объекта и закрывают свои дескрипторы этого объекта, диспетчер объектов уменьшает показатель счетчика открытых дескрипторов. Когда показания счетчика падают до нуля, диспетчер объектов удаляет имя объекта из своего глобального пространства имен. Это удаление не дает процессам открыть дескриптор объекта.

Второй этап сохранения объектов заключается в прекращении сохранения самих объектов (то есть в их удалении), когда они больше не используются. Поскольку код операционной системы обычно обращается к объектам путем использования указателей, а не дескрипторов, диспетчер объектов должен также записывать количество указателей на объект, розданных им системным процессам. Он увеличивает показатель счетчика ссылок на объект при каждой выдаче указателя на этот объект; когда компоненты режима ядра завершают использование указателя, они вызывают диспетчер объекта для уменьшения показателя счетчика ссылок на объект. Система также увеличивает показатель счетчика ссылок при увеличении показателя счетчика дескрипторов, и по аналогии с этим уменьшает показатель счетчика ссылок при уменьшении показателя счетчика дескрипторов, поскольку дескриптор также является отслеживаемой ссылкой на объект.

На рис. 3.23 показаны два используемых объекта событий. Первое событие открыто процессом А. Процессом Б открыты оба события. Кроме того, на первое событие сослалась некая структура режима ядра; стало быть, показатель счетчика ссылок равен 3. Следовательно, если процессы А и Б закрывают свои дескрипторы, относящиеся к первому объекту события, он все равно продолжит свое существование, потому что показатель его счетчика ссылок равен 1. Но, когда процесс Б закрывает свой дескриптор, относящийся к второму объекту события, этот объект будет освобожден.

Следовательно, даже после того как счетчик открытых дескрипторов дойдет до нуля, у счетчика ссылок на объект может оставаться положительное значение, свидетельствующее о том, что операционная система по-прежнему использует объект. В итоге, когда счетчик ссылок дойдет до нуля, диспетчер объектов удалит объект из памяти. Это удаление должно отвечать определенным правилам, кроме того, оно требует в некоторых случаях взаимодействия с вызывающей программой. Например, из-за того что объекты могут присутствовать как в выгружаемом, так и в невыгружаемом пуле памяти, если удаление ссылки случится на IRQL-уровне диспетчера или на более высоком уровне, оно приведет к тому, что показатель счетчика ссылок упадет до нуля, и при попытке немедленно освободить память объекта, существующего в выгружаемом пуле памяти, система

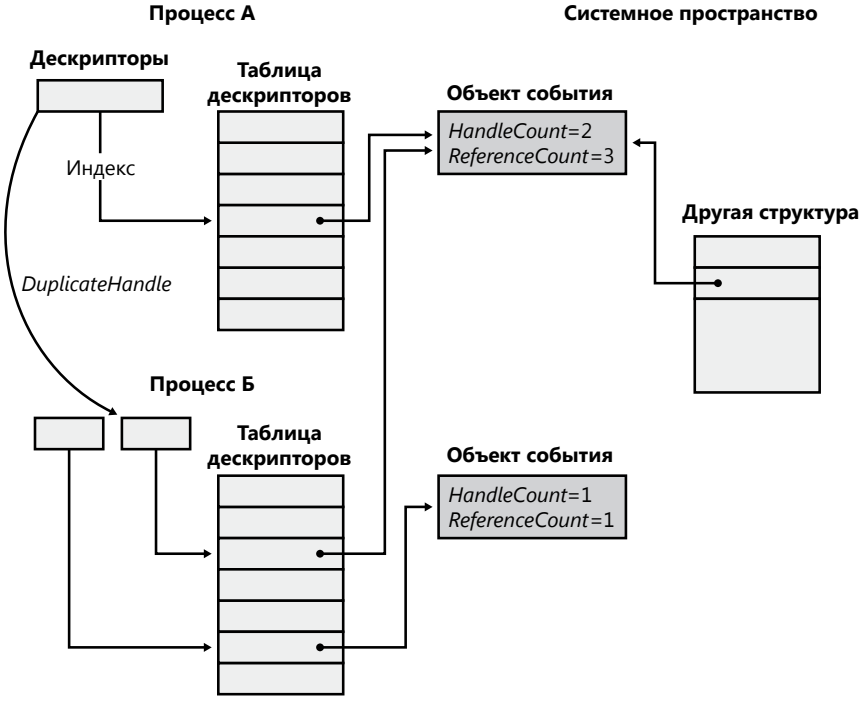


Рис. 3.23. Счетчики дескрипторов и ссылок

войдет в аварийное состояние[1]. При таком развитии событий диспетчер объектов выполняет отложенную операцию удаления, ставя операцию в очередь рабочего потока, запущенного на уровне passive (IRQL 0). В этой главе к системным рабочим потокам мы еще вернемся.

Еще один сценарий, требующий отложенного удаления, касается работы с объектами диспетчера транзакций ядра — Kernel Transaction Manager (KTM). Могут сложиться такие обстоятельства, при которых конкретные драйверы будут удерживать блокировку, связанную с тем или иным объектом, и попытка удаления такого объекта выльется в попытку системы получить эту блокировку. Но драйвер может так и не получить шанса на ее освобождение, вызывая тем самым взаимную блокировку. При работе с KTM-объектами разработчики драйверов должны, не обращая внимания на IRQL-уровень, использовать для принудительного отложенного удаления функцию ObDereferenceObjectDeferDelete. И наконец, этот же механизм используется также диспетчером ввода-вывода для оптимизации своей работы, чтобы конкретная операция ввода-вывода могла завершиться быстрее и не ждать, пока диспетчер объектов удалит объект.

Благодаря тому способу, которым осуществляется сохранение объектов, приложение может обеспечить наличие в памяти объекта и его имени, просто сохраняя открытым дескриптор объекта. Программисты, создающие приложения, которые содержат два и более взаимодействующих процессов, могут не беспо-

[1] Следует напомнить, что такое обращение будет запрещенным, поскольку ошибка обращения к странице никогда не будет обслужена.

коиться о том, что один процесс может удалить объект еще до того, как другой процесс не завершит его использование. Кроме того, закрытие дескрипторов объекта, принадлежащих приложениям, не приведет к удалению объекта, если он все еще используется операционной системой. Например, один процесс может создать второй процесс для выполнения программы в фоновом режиме, затем этот процесс сразу же закрывает свой дескриптор на второй процесс. Поскольку второй процесс нужен операционной системе для выполнения программы, она сохраняет ссылку на этот объект процесса. И только когда фоновая программа завершит свое выполнение, диспетчер объектов уменьшит показатель счетчика ссылок на второй процесс, а затем удалит его объект.

Утечки памяти, связанные с объектами, могут представлять опасность для системы, вызывая утечки пула памяти ядра и, в конечном итоге, вызывая истощение общесистемной памяти (может прекратить работу приложений с трудно выявляемыми причинами). Поэтому в Windows имеется ряд отладочных механизмов, которые могут быть включены для отслеживания, анализа и отладки проблемного кода, связанного с использованием дескрипторов и объектов. Кроме того, в Debugging Tools for Windows имеются еще два расширения, которые подключаются к этим механизмам и предоставляют удобное средство для проведения графического анализа. Описание этих механизмов дано в табл. 3.16.

Таблица 3.16. Механизмы отладки для дескрипторов объектов

Механизм	Чем включается	Расширение отладчика ядра
Handle Tracing Database (База данных отслеживания дескрипторов)	Общесистемным средством отслеживания стека ядра — Kernel Stack Trace и (или) установкой для каждого процесса с помощью средства Gflags.exe флага отслеживания пользовательского стека — User Stack Trace	!htrace <значение дескриптора > < ID процесса>
Object Reference Tracing (Отслеживание ссылок на объект)	Установка в режиме отслеживания ссылок на объект — Object Reference Tracing, флагов для имени конкретного процесса (или имен конкретных процессов) или флагов тега (тегов) пула конкретного типа объекта с помощью средства Gflags.exe	!obtrace <указатель на объект >
Object Reference Tagging (Тегирование ссылок на объект)	Соответствующие API-функции должны вызываться драйверами	Не определено

Включение базы данных отслеживания дескрипторов пригодится при попытке разобраться с использованием каждого описателя внутри контекста приложения или системы. Расширение отладчика !htrace может показать моментальный снимок трассировки стека на время открытия конкретного дескриптора. После обнаружения утечки памяти, связанной с дескриптором, трассировка стека может точно определить код, создающий дескриптор, который может быть проанализирован на предмет пропущенного вызова такой функции, как CloseHandle.

Расширение, позволяющее отслеживать ссылки на объект — !obtrace, — позволяет вести более широкое наблюдение, показывая трассировку стека для каждого

вновь созданного дескриптора, а также при каждой ссылке на дескриптор из ядра (а также при каждом ее открытии, дублировании или наследовании) и при каждом удалении ссылки. При каждом анализе подобных схем появляется возможность более легкой отладки кода при неправильном использовании объекта на системном уровне. Кроме того, такое отслеживание ссылок дает возможность разобраться в поведении системы при работе с конкретными объектами. К примеру, при отслеживании процессов выводятся ссылки от всех, имеющихся в системе драйверов, которые зарегистрировали уведомительные функции обратного вызова (как, например, Process Monitor). Это помогает выявить неконтролируемые или некачественные драйверы сторонних производителей, которые могут ссылаться на дескрипторы в режиме ядра, никогда не удаляя ссылок на них.

В отличие от предыдущих двух механизмов, тегирование ссылок на объект не является функцией отладки, которую можно включить с помощью глобальных флагов или отладчика, а представляет собой набор API-функций, которые должны использоваться разработчиками драйверов устройств для установки и удаления ссылок на объекты, включающий такие функции, как ObReferenceObjectWithTag и ObDereferenceObjectWithTag. Так же как и при тегировании пулов, эти API-функции позволяют разработчикам предоставлять каждой паре установки-удаления ссылки для ее идентификации тег, состоящий из четырех символов. При использовании только что рассмотренного расширения !obtrace показывается также и тег для каждой операции установки и удаления ссылки, что позволяет использовать в качестве механизма идентификации мест утечки памяти или подчиненных ссылок не только стек вызовов, это особенно важно, когда заданный вызов осуществляется драйвером тысячи раз.

ПРИМЕЧАНИЕ

При включении отслеживания ссылок на объект для конкретного типа объекта можно получить имя его тега пула, просматривая элемент «key» структуры OBJECT_TYPE при использовании команды dt. У каждого объекта типа в системе есть глобальная переменная со ссылкой на эту структуру, например PsProcessType. Кроме этого можно воспользоваться командой !object, которая показывает указатель на структуру.

Учет ресурсов

Учет ресурсов, как и сохранение объектов, тесно связан с использованием дескрипторов объектов. Положительный показатель счетчика дескрипторов свидетельствует о том, что данным ресурсом пользуется какой-то процесс. Он также свидетельствует о том, что какой-то процесс несет ответственность за память, занимаемую объектом. Когда счетчики дескрипторов и ссылок сбрасываются в нуль, тот процесс, который использовал объект, больше не должен нести ответственность за этот объект.

Чтобы наложить ограничения на доступ к ресурсам со стороны процесса, во многих операционных системах используется система квот. Но типы квот, налагаемых на процессы, порой отличаются друг от друга, усложняя ситуацию, и код, отслеживающий квоты, распространяется по всей операционной системе. Например, в некоторых операционных системах компонент ввода-вывода может записывать и ограничивать количество файлов, открываемых процессом, а ком-

понент памяти может накладывать лимит на объем памяти, распределяемый потокам процесса. Компонент процесса может ограничивать пользователя в количестве новых, создаваемых процессов или в количестве потоков внутри процесса. Каждое из таких ограничений отслеживается и приводится в исполнение в различных частях операционной системы.

В отличие от этого, диспетчер объектов Windows предоставляет централизованную возможность учета ресурсов. В каждом заголовке объекта есть атрибут под названием *квота* (quota charges), в котором записывается, сколько диспетчером объектов вычитается ресурсов из выделенной процессу квоты пулов выгружаемой и (или) невыгружаемой памяти, когда поток в процессе открывает дескриптор объекта.

Каждый процесс в Windows указывает на структуру квот, в которой записываются лимиты и текущее значение используемых невыгружаемого пула, выгружаемого пула и файла подкачки. По умолчанию эти квоты имеют значение 0 (без ограничений), но могут быть указаны путем изменения значений реестра. Для этого нужно дополнить или отредактировать параметры NonPagedPoolQuota, PagedPoolQuota и PagingFileQuota в разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\MemoryManagement. Следует заметить, что все процессы в интерактивном сеансе работы используют общий блок квот, и документированного способа создания процессов со своими собственными блоками квот не существует.

Имена объектов

Важным моментом в создании множества объектов является потребность в разработке четкой системы для их отслеживания. Чтобы помочь в решении этой задачи, диспетчеру объектов требуется следующая информация:

- способ, позволяющий отличить один объект от другого;
- метод для поиска и извлечения конкретного объекта.

Первое требование выполняется путем предоставления возможности назначать объектам имена. Это расширение предоставляется большинством операционных систем и заключается, к примеру, в возможности давать имена отдельным ресурсам, файлам, каналам или блокам общей памяти. Исполняющая система, в отличие от этого, позволяет любому ресурсу, чтобы у него было имя, быть представленным в виде объекта. Второе требование, поиск и извлечение объекта также удовлетворяется с помощью имен объектов. Если диспетчер объектов хранит объекты по именам, он может найти объект путем поиска его имени.

Имена объектов также удовлетворяют третьему требованию, которое заключается в разрешении процессам совместно пользоваться объектами. Пространство имен объектов исполняющей системы является глобальным, видимым всем процессам системы. Один процесс может создать объект и пометить его имя в глобальное пространство имен, а второй процесс может открыть дескриптор объекта, указав его имя. Если объект не предназначен для подобного совместного использования, его создатель не нуждается в предоставлении имени этому объекту.

Для повышения эффективности работы диспетчер объектов не разыскивает имя объекта всякий раз, когда кто-нибудь использует объект. Он ищет имя только при двух обстоятельствах. Во-первых, когда процесс создает объект с именем: диспетчер объектов ищет это имя перед сохранением нового имени в глобальном

пространстве имен, чтобы проверить, что такого имени еще не существует. Во-вторых, когда процесс открывает дескриптор объекта с именем: диспетчер объектов ищет имя, находит по нему объект, а затем возвращает дескриптор объекта вызвавшей его программе; после этого вызывающая программа использует для ссылки на объект его дескриптор. При поиске имени диспетчер объектов разрешает вызывающей программе выбрать либо поиск с учетом регистра символов, либо поиск без его учета, то есть выбрать функцию, поддерживаемую подсистемой для приложений UNIX и другими средами окружения, использующими имена файлов, чувствительные к регистру символов.

Каталоги объектов

Объект каталога объектов является средством диспетчера объектов, предназначенным для поддержки этой иерархической структуры присвоения имен. Этот объект аналогичен каталогу файловой системы и содержит имена других объектов и даже, возможно, других каталогов объектов. Объект каталога объектов сохраняет достаточный объем информации для перевода этих имен объектов в указатели на сами объекты. Диспетчер объектов использует указатели для создания дескрипторов объектов, которые он возвращает вызывающим программам пользовательского режима. Каталоги объектов, в которых диспетчер хранит объекты, может создавать как код режима ядра (включая компоненты исполняющей системы и драйверы устройств), так и код пользовательского режима (например, код подсистем). Например, диспетчер ввода-вывода создает каталог объектов \Device, который содержит имена объектов, представляющих устройства ввода-вывода.

Место хранения имен объектов зависит от типа объекта. В табл. 3.17 перечислены стандартные каталоги объектов, находящиеся во всех системах Windows, и те типы объектов, чьи имена в них хранятся. Из перечисленных каталогов стандартные Windows-приложения видят только каталоги \BaseNamedObjects и \Global?? (см. раздел «Пространство имен сеанса»).

Таблица 3.17. Каталоги стандартных объектов

Каталог	Типы хранящихся в нем объектов
\ArcName	Символические ссылки, отображающие пути в ARC-стиле на пути в NT-стиле
\BaseNamed-Objects	Объекты глобальных мьютексов, событий, семафоров, таймеров ожидания, заданий, ALPC-портов, символических ссылок и разделов
\Callback	Объекты обратных вызовов
\Device	Объекты устройств
\Driver	Объекты драйверов
\FileSystem	Объекты драйверов файловой системы и объекты устройства распознавания файловой системы. Диспетчер фильтров (Filter Manager) также создает свой собственный диспетчер объектов в подразделе Filters
\GLOBAL??	Имена устройств MS-DOS. (Каталоги \Sessions\0\DosDevices\<*LUID*>\Global являются символическими ссылками на этот каталог.)
\Kernel-Objects	Содержит объекты событий, сигнализирующих об условиях дефицита ресурсов, ошибках памяти, завершении конкретных задач операционной системы, а также объекты, представляющие сеансы (Sessions)

Каталог	Типы хранящихся в нем объектов
\KnownDlls	Имена разделов и пути для известных DLL-библиотек (DLL-библиотеки отображаются в системе во время запуска)
\KnownDlls32	На установке 64-разрядной Windows \KnownDlls содержит присущие ей 64-разрядные двоичные коды, поэтому этот каталог используется вместо него для хранения Wow64 32-разрядные версии этих DLL-библиотек
\Nls	Имена разделов для отображения таблиц поддержки национального языка
\ObjectTypes	Имена типов объектов
\PSXSS	Если включена работа подсистемы для приложений UNIX (посредством установки компонента SUA), этот каталог содержит ALPC-порты, используемые этой подсистемой
\RPC Control	ALPC-порты, используемые удаленными вызовами процедур (RPC-вызовами), и события, используемые Conhost.exe в качестве части механизма изоляции консоли
\Security	ALPC-порты и события, используемые именами объектов, определенных для подсистемы безопасности
\Sessions	Каталог пространства имен, относящегося к сеансу. (См. следующий подраздел.)
\UMDFCommunicationPorts	ALPC-порты, используемые средой драйверов пользовательского режима — User-Mode Driver Framework (UMDF)
\Windows	ALPC-порты подсистемы Windows, совместно используемые разделы и объекты window station

Поскольку имена основных объектов ядра, таких как мьютексы, события, семафоры, таймеры ожидания и разделы, хранятся в едином каталоге объектов, никакие два объекта этих типов не могут иметь одно и то же имя, даже если они относятся к объектам разных типов. Это ограничение подчеркивает необходимость тщательного выбора имен, чтобы они не конфликтовали с другими именами. Например, можно использовать в качестве префикса имени глобально уникальный идентификатор (GUID) и (или) скомбинировать имя с идентификатором безопасности пользователя (SID).

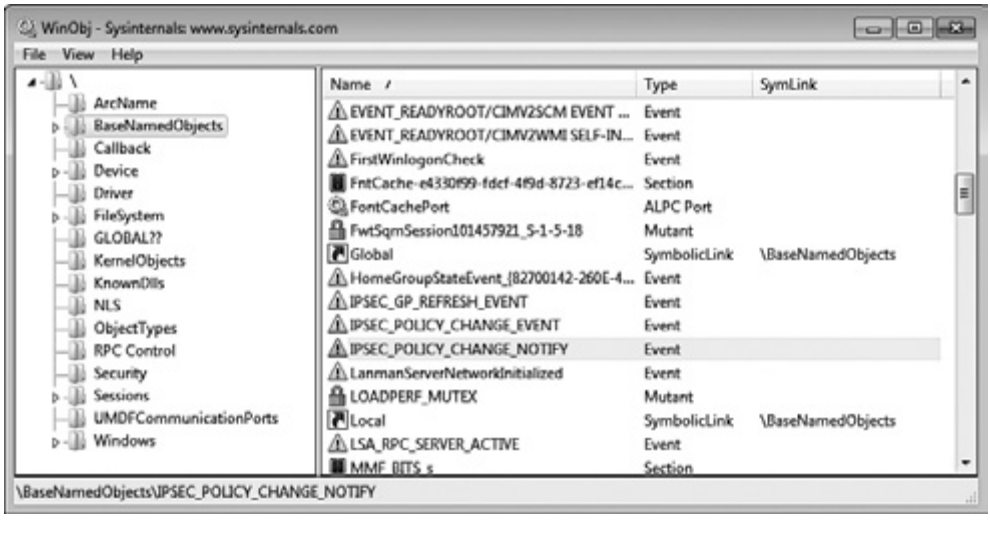
Имена объектов являются глобальными для отдельного компьютера (или для всех процессов на мультипроцессорном компьютере), но они не видимы в сети. Тем не менее имеющийся в диспетчере объектов метод анализа позволяет получать доступ к обладающим именами объектам, существующим на других компьютерах. Например, диспетчер ввода-вывода, предоставляющий службы для работы с файловыми объектами, расширяет функции диспетчера объектов на удаленные файлы. При запросе на открытие удаленного файлового объекта диспетчер объектов вызывает метод анализа, позволяющий диспетчеру ввода-вывода перехватить запрос и доставить его к сетевому устройству перенаправления, драйверу, обращающемуся к файлам по сети. Серверный код на удаленной системе Windows вызывает диспетчер объектов и диспетчер ввода-вывода на этой системе для поиска файлового объекта и возвращает информацию обратно по сети.

При работе с обладающими именами объектами нужно руководствоваться некоторыми соображениями безопасности, связанными с возможностью сквоттинга, то есть незаконного присвоения имени объекта. Хотя объекты в различных сеансах защищены друг от друга, внутри пространства имен текущего сеанса стандартной защиты, которая могла бы быть установлена с помощью стандартных API-функций Windows, не существует. Это дает возможность непривилегированным приложениям запускаться в одном сеансе с привилегированными приложениями для доступа к их объектам, как было рассмотрено ранее в подразделе, посвященном безопасности объектов. К сожалению, даже если создатель объекта использует для защиты объекта правильный DACL-список, это не спасает от сквоттинг-атаки, при которой непривилегированное приложение создает объект до привилегированного приложения, отказывая таким образом в доступе законному приложению.

Чтобы смягчить данную проблему, Windows выставляет концепцию защищенного пространства имен. Она позволяет приложениям пользовательского режима создавать каталоги объектов с помощью API-функции CreatePrivateNamespace и связывать эти каталоги с дескрипторами границ, являющимися специальными структурами данных, защищающими каталоги. Эти дескрипторы содержат SID-идентификаторы, описывающие, каким принципалам безопасности разрешен доступ к каталогу объектов. Таким образом, привилегированное приложение может быть уверено, что непривилегированные приложения не смогут проводить атаку отказа от обслуживания в отношении этих объектов, что не остановит привилегированные приложения от совершения таких же действий (но это спорный момент). Кроме того, дескриптор границы может также содержать уровень целостности, основанный на уровне целостности процесса, защищая объекты, которые, возможно, принадлежат той же самой учетной записи пользователя, что и приложение (см. главу 6).

ЭКСПЕРИМЕНТ: ПРОСМОТР ОСНОВНЫХ ОБЪЕКТОВ С ИМЕНАМИ

Список основных объектов, обладающих именами, можно просмотреть с помощью средства WinObj от Sysinternals. Запустите Winobj.exe и щелкните на каталоге \BaseNamedObjects.



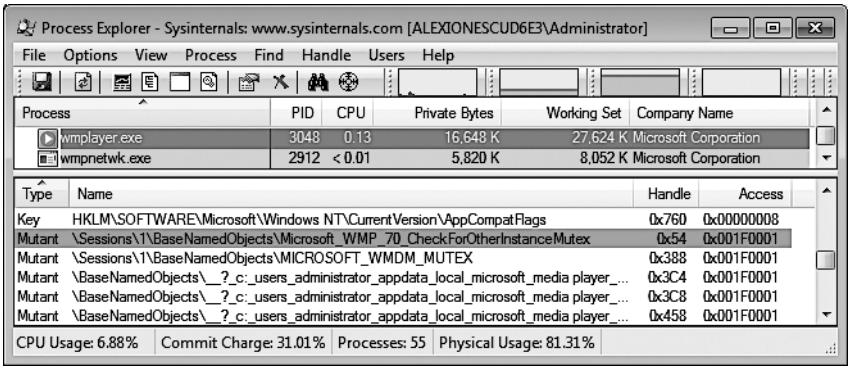
Объекты с именами показаны справа. Значки показывают тип объекта:

- Мьютексы обозначены символом замка.
- Разделы (Windows-объекты, отображенные на файлы) показаны как микросхемы памяти.
- События показаны как восклицательные знаки.
- Семафоры показаны значком, напоминающим светофор.
- Символические ссылки сопровождаются значками изогнутых стрелок.
- Папками обозначены объекты каталогов.
- Зубчатыми шестернями обозначены другие объекты, например ALPC-порты. ■

ЭКСПЕРИМЕНТ: САМОВОЛЬНОЕ ВМЕШАТЕЛЬСТВО В ПРОЦЕСС ИСПОЛЬЗОВАНИЯ ЕДИНСТВЕННОГО ЭКЗЕМПЛЯРА

Такие приложения, как Windows Media Player и приложения, входящие в состав Microsoft Office, могут послужить ярким примером принудительного использования единственного экземпляра посредством именованных объектов. Следует заметить, что при запуске исполняемого файла Wmplayer.exe Windows Media Player появляется только один раз, каждый последующий запуск приводит лишь к тому, что окну приложения возвращается фокус. Вы можете вмешаться в список дескрипторов с помощью Process Explorer, превратить компьютер в медиа-микшер! Выполните следующие действия:

1. Запустите Windows Media Player и Process Explorer, чтобы просмотреть таблицу дескрипторов (щелкнув для этого на пунктах View (Вид), Lower Pane View (Просмотр нижней панели), а затем на пункте Handles (Дескрипторы)). Вы увидите дескриптор, в столбце имени которого содержится строка CheckForOtherInstanceMutex.



2. Щелкните правой кнопкой мыши на дескрипторе и выберите пункт Close Handle (Закрыть дескриптор). Подтвердите действие при запросе.
3. Теперь запустите Windows Media Player еще раз. Обратите внимание на то, что теперь будет создан второй процесс.
4. А теперь проиграйте в каждом экземпляре разные песни. Можно также воспользоваться Микшером в Области уведомлений (щелкнув для этого на значке Динамики), чтобы выбрать, у какого из двух процессов будет более высокий уровень громкости, создав при этом эффективно работающую среду микширования звука.

Вместо закрытия дескриптора именованного объекта, до Windows Media Player могло бы быть запущено какое-нибудь другое приложение, создающее объект с таким же именем. При таком развитии событий Windows Media Player никогда бы не запустился, ошибочно полагая, что он уже запущен в системе. ■

Символические ссылки (symbolic links). В некоторых файловых системах (например, в NTFS и в некоторых UNIX-системах) символическая ссылка дает возможность пользователю создать имя файла или имя каталога, при использовании которого этот имя превращается операционной системой в другое имя файла или каталога. Использование символической ссылки является простым методом, позволяющим пользователям опосредованно организовывать совместное использование файла или содержимого каталога, создавая перекрестную связь между различными каталогами в обычной иерархической структуре каталогов.

Диспетчер объектов реализует объект, который называется *объектом символической ссылки*, и выполняет аналогичные функции для имен объектов в своем пространстве имен объектов. Символическая ссылка может встречаться в любом месте строки с именем объекта. Когда вызывающая программа обращается к имени объекта символической ссылки, диспетчер объектов проходит по своему пространству имен объектов, пока ему не встретится объект символической ссылки. Он заглядывает в символическую ссылку и находит строку, которую заменяет именем, на которое указывала символическая ссылка. Затем он перезапускает свою функцию поиска имени.

Одним из мест, где исполняющая система использует объекты символических ссылок, является трансляция имен устройств в стиле MS-DOS во внутренние имена устройств Windows. В Windows пользователь обращается к жесткому диску, используя имена C:, D: и т. д., а при обращении к последовательным портам он использует имена COM1, COM2 и т. д. Подсистема Windows делает из этих объектов символических ссылок защищенные, глобальные данные, помещая их в каталог \Global?? пространства имен диспетчера объектов.

Пространство имен сеанса

У служб есть доступ к *глобальному* пространству имен, которое служит в качестве первого экземпляра пространства имен. Дополнительным сеансам дается закрытый, предназначенный для конкретного сеанса вид пространства имен, известный как *локальное* пространство имен. Часть пространства имен, локализированная для каждого сеанса, включает каталоги \DosDevices, \Windows и \BaseNamedObjects. Создание отдельных копий одной и той же части пространства имен известно как *создание экземпляров* пространства имен. Создание экземпляра \DosDevices дает каждому пользователю возможность иметь другие буквы сетевых дисков и таких объектов Windows, как последовательные порты. В Windows глобальный каталог \DosDevices называется \Global?? и является каталогом, на который указывает \DosDevices, а локальные каталоги \DosDevices идентифицируются с помощью идентификатора входа в систему того или иного сеанса (logon session ID).

В каталог \Windows драйвер Win32k.sys вставляет создаваемый Winlogon, интерактивный объект станции окна (window station) — \WinSta0. Службы терминалов (Службы удаленных рабочих столов) могут поддерживать несколько интерактивных пользователей, но каждому пользователю нужна индивидуальная

версия WinSta0, чтобы сохранить иллюзию обращения к предопределенной интерактивной станции окна в Windows. И наконец, приложения и система создают в каталоге \BaseNamedObjects объекты совместного использования, куда включаются события, мьютексы и разделы памяти. Если двое пользователей запускают приложение, создающее объект с именем, у каждого пользовательского сеанса должна быть закрытая версия объекта, чтобы эти два экземпляра приложения не мешали друг другу, обращаясь к одному и тому же объекту.

Диспетчер объектов реализует локальное пространство имен путем создания закрытых версий трех каталогов, указываемых ниже каталога, связанного с пользовательским сеансом, ниже \Sessions*n* (где *n* — идентификатор сеанса). Когда приложение Windows в удаленном сеансе номер два создает, к примеру, событие с именем, диспетчер объектов явным образом перенаправляет имя объекта с \BaseNamedObjects на \Sessions\2\BaseNamedObjects. Все функции диспетчера объектов, связанные с управлением пространством имен в курсе наличия нескольких экземпляров каталогов и участвуют в создании иллюзии того, что все сеансы используют одно и то же пространство имен. DLL-библиотеки подсистемы Windows предваряют те имена, передаваемые приложениями Windows, которые ссылаются на объекты \DosDevices, префиксами \?? (например, C:\Windows становится \??\C:\Windows). Когда диспетчер объектов видит специальный префикс \??, предпринимаемые им действия зависят от версии Windows, но он всегда полагается на поле DeviceMap в объекте процесса исполняющей системы (EPROCESS будет рассмотрен в главе 5), которое указывает на структуру данных, совместно используемую другими процессами одного и того же сеанса.

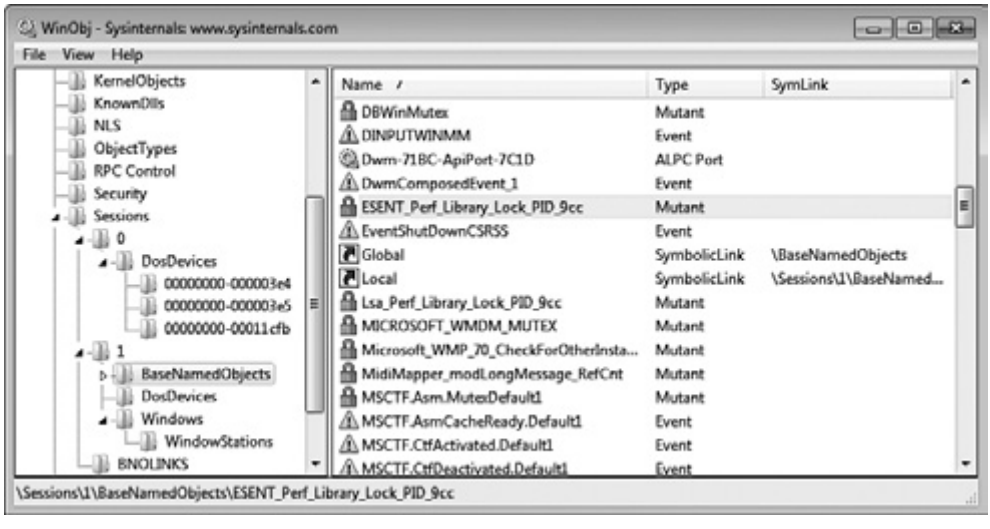
Поле DosDevicesDirectory структуры DeviceMap указывает на каталог диспетчера объектов, который представляет \DosDevices локального процесса. Когда диспетчер объектов видит ссылку на \??, он определяет местонахождение \DosDevices локального процесса, используя поле DosDevicesDirectory структуры DeviceMap. Если диспетчер объектов не находит объект в каталоге, он проверяет поле DeviceMap объекта каталога. Если это поле имеет значение, он ищет объект в каталоге, на который указывает поле GlobalDosDevicesDirectory структуры DeviceMap, в качестве которого всегда выступает каталог \Global??.

При определенных обстоятельствах приложения, осведомленные о сеансе, нуждаются в доступе к объектам, принадлежащим глобальному сеансу, даже если приложение запущено в другом сеансе. Приложению это может понадобиться для синхронизации со своими же экземплярами, запущенными в других удаленных сеансах или с помощью сеанса консоли (то есть сеанса 0). Для таких случаев диспетчер объектов предоставляет специальную подмену \Global, которую приложение может использовать в качестве префикса к любому имени объекта для доступа к глобальному пространству имен. Например, приложение, запущенное в сеансе номер два, открывающее объект \Global\ApplicationInitialized, направляется на \BaseNamedObjects\ApplicationInitialized, а не на \Sessions\2\BaseNamedObjects\ApplicationInitialized.

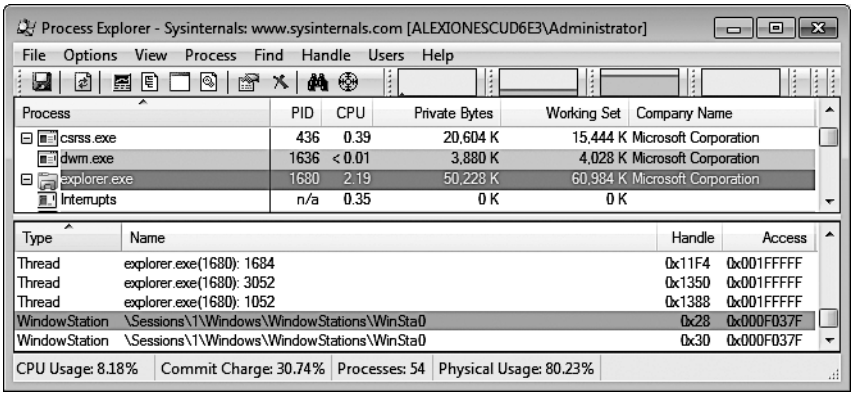
Приложение, которому нужен доступ к объекту в глобальном каталоге \DosDevices, не нуждается в использовании префикса \Global, пока такого объекта нет в его локальном каталоге \DosDevices. Причина в том, что диспетчер объектов автоматически ищет объект в глобальном каталоге, если не находит его в локальном каталоге. Но приложение может заставить проводить проверку в глобальном каталоге, используя ключ \GLOBALROOT.

ЭКСПЕРИМЕНТ: ПРОСМОТР ЭКЗЕМПЛЯРОВ ПРОСТРАНСТВА ИМЕН

Разделение пространства имен сеанса 0 от пространств имен других сеансов можно увидеть сразу же после входа в систему. Дело в том, что первый пользователь консоли, входящий в систему, попадает в сеанс 1 (а службы запускаются в сеансе 0). Запустите средство Winobj.exe и щелкните на каталоге \Sessions. Вы увидите подкаталог с цифровым именем для каждого активного сеанса. Если открыть один из таких каталогов, можно будет увидеть подкаталоги с именами \DosDevices, \Windows и \BaseNamedObjects, которые являются подкаталогами локального пространства имен, принадлежащего сеансу. Локальное пространство имен показано на следующей копии экрана.



Затем запустите Process Explorer и выберите какой-нибудь процесс в своем сеансе (например, Explorer.exe), а затем просмотрите таблицу дескрипторов (View (Вид)▸ Lower Pane View (Просмотр нижней панели)▸ Handles (Дескрипторы)). Вы увидите дескриптор \Windows\WindowStations\WinSta0 ниже \Sessions*n*, где *n* является идентификатором сеанса (session ID).



Каталоги сеансов изолированы друг от друга, и для создания глобального объекта (за исключением объектов разделов) требуются административные привилегии. Перед тем как разрешить проведение подобных операций, проверяется наличие специальной привилегии под названием *создание глобального объекта* (create global object).

Фильтрация объектов

Windows включает в диспетчер объектов модель фильтрации. Одним из основных положительных свойств этой модели фильтрации является возможность использования понятия *высоты* (altitude), которое применяется в данных существующих технологиях фильтрования, это означает, что несколько драйверов могут фильтровать события диспетчера объектов в соответствующих местах стека фильтрации. Кроме того, драйверам разрешается перехватывать вызовы таких функций, как NtOpenThread и NtOpenProcess, и даже изменять маски доступа, запрошенные из диспетчера процессов. Это позволяет защититься от конкретных операций над открытым дескриптором, но операция открытия не может быть полностью заблокирована, потому что это слишком бы напомнило злонамеренное действие (процессы, не поддающиеся никакому управлению).

Более того, драйверы получают возможность воспользоваться как предшествующими, так и последующими обратными вызовами, что позволит им приготовиться к конкретной операции до ее осуществления, а также отреагировать на информацию или придать ей окончательную форму после завершения этой операции. Эти обратные вызовы могут быть указаны для каждой операции (на данный момент поддерживаются только операции создания, открытия и создания дубликата) и могут быть указаны для каждого типа объекта (на данный момент поддерживаются только объект процесса и объект потока). Для каждого обратного вызова драйверы могут указать свое собственное внутреннее значение контекста, который может возвращаться при всех вызовах драйвера или при вызовах пары предваряющего и завершающего обратных вызовов. Эти обратные вызовы могут быть зарегистрированы с помощью API-функции ObRegisterCallbacks, а их регистрация может быть отменена с помощью API-функции ObUnregisterCallbacks, причем отмена регистрации возлагается на драйвер.

Использование API-функций ограничено образами, имеющими следующие характеристики:

- Образ должен иметь подпись, даже на 32-разрядных компьютерах, в соответствии с теми же правилами, которые сформулированы в политике подписи кода режима ядра — Kernel Mode Code Signing (KMCS). Образ должен быть скомпилирован с установленным ключом компоновщика /integritycheck, который приводит к установке значения IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY в PE-заголовке. Это заставляет диспетчер памяти проверять подпись образа независимо от любых других установок по умолчанию, которые обычно могут не приводить к проверке.
- Образ должен быть подписан с каталогом, содержащим криптографические хэши для каждой страницы исполняемого кода. Это позволяет системе обнаружить изменения образа после его загрузки в память.

Перед выполнением обратного вызова диспетчер объектов вызывает функцию MmVerifyCallbackFunction в отношении указателя на целевую функцию, которая, в свою очередь, определяет местонахождение таблицы данных загрузчика, связанной с модулем, владеющим этим адресом, и проверяет, установлен или нет флаг LDRP_IMAGE_INTEGRITY_FORCED (см. раздел «База данных загруженных модулей»).

Синхронизация

При разработке операционных систем важную роль играет понятие взаимного исключения. Оно относится к обеспечению того, что в каждый момент времени доступ к конкретному ресурсу может иметь один и только один поток. Взаимное исключение необходимо, когда ресурс не предполагает совместного использования или когда такое использование может привести к непредсказуемому исходу. Например, если два потока одновременно копируют файл на порт принтера, их вывод может быть перепутан. Аналогично этому, если один поток производит чтение из какого-нибудь места памяти, в то время как другой поток ведет запись в это же место, первый поток получит непредсказуемые данные. В общем, ресурсы, которые могут подвергаться записи, не могут совместно использоваться без ограничений, а ресурсы, которые не подвергаются изменениям, могут использоваться совместно. На рис. 3.24 показано, что произойдет, когда и тот и другой из двух потоков, запущенные на разных процессорах, записывают данные в круговую очередь.

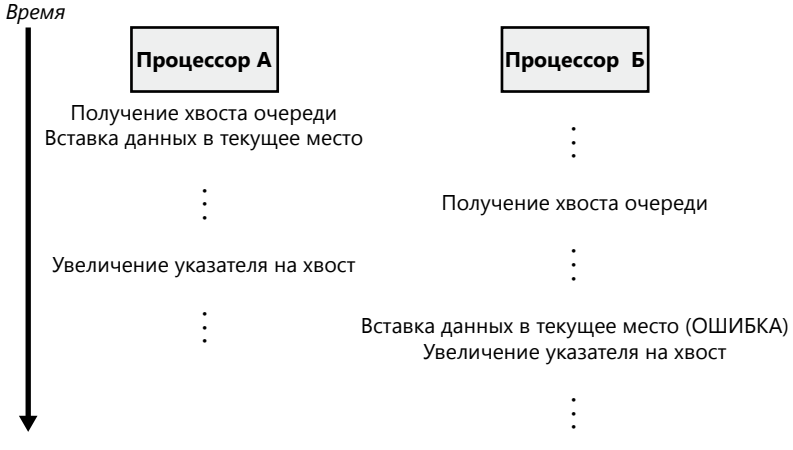


Рис. 3.24. Неправильное совместное использование памяти

Поскольку второй поток получает значение указателя на хвост очереди до того, как первый поток закончит его обновление, второй поток вставляет свои данные в то же самое место, которое было использовано первым потоком, перезаписывая данные и отставляя пустое место в очереди. Хотя на рис. 3.24 показано то, что может произойти на мультипроцессорной системе, такая же ошибка может произойти и на однопроцессорной системе, если операционная система осуществляет переключение контекста на второй поток перед тем, как первый поток обновит указатель на хвост очереди.

Разделы кода, обращающиеся к необобщаемому ресурсу, называются *критическими* разделами. Чтобы получился правильный код, нужно обеспечить, чтобы критический раздел в отдельно взятый момент времени мог выполняться только одним потоком. Пока один поток ведет запись в файл, обновляя базу данных или изменяя значение общей переменной, никакой другой поток не должен получать разрешение на доступ к тому же самому ресурсу. Псевдокод, показанный на рис. 3.24, является критическим разделом, который, вопреки правилам, без взаимного исключения получил доступ к совместно используемой структуре данных.

Хотя вопрос использования взаимного исключения важен для всех операционных систем, он особенно важен (и весьма сложен в решении) для *тесно связанных, симметричных многопроцессорных* (symmetric multiprocessing, SMP) операционных систем типа Windows, в которых один и тот же системный код запускается одновременно на более чем одном процессоре, совместно используя определенные структуры данных, хранящиеся в глобальной памяти. В Windows предоставление механизмов, которые могут использоваться системным кодом для предотвращения ситуаций, при которых два потока могли бы одновременно изменять одну и ту же структуру данных, возложена на ядро. Оно предоставляет примитивы взаимного исключения, используемые им самим и остальной исполняющей системой для синхронизации доступа к глобальным структурам данных.

Поскольку планировщик синхронизирует доступ к своим структурам данных на IRQL-уровне DPC/dispatch, ядро и исполняющая система не могут полагаться на механизмы синхронизации, которые будут приводить к ошибкам обращения к странице или операциям перепланирования для синхронизации доступа к структурами данных, когда IRQL находится на уровне DPC/dispatch или выше (на уровнях, известных как повышенный или высокий IRQL). В следующих разделах вы узнаете, как ядро и исполняющая система используют взаимное исключение для защиты своих глобальных структур данных при высоком IRQL, и какие механизмы взаимных исключений и синхронизации ядро и исполняющая система используют при низком IRQL (ниже уровня DPC/dispatch).

Высокоуровневая IRQL-синхронизация

На разных стадиях своей работы ядро должно гарантировать, что один и только один процессор в каждый отдельно взятый момент времени выполняет критический раздел. Критическими разделами ядра являются сегменты кода, изменяющие глобальную структуру данных, такую как базу данных диспетчера ядра или его DPC-очередь. Операционная система не может корректно работать, пока ядро не сможет гарантировать, что потоки обращаются к этим структурам данных с применением метода взаимного исключения.

Самой большой областью озабоченности являются прерывания. Например, при возникновении прерывания ядро может находиться в процессе обновления глобальной структуры данных, а процедура обработки прерывания также занимается обновлением структуры. Простые однопроцессорные операционные системы иногда предотвращают такой сценарий, запрещая все прерывания при каждом доступе к глобальным данным, но ядро Windows стоит перед более сложным решением. Перед использованием глобального ресурса ядро временно маскирует те прерывания, чьи обработчики также используют этот ресурс. Оно делает это путем подъема

IRQL процессора на наивысший уровень, используемый любым потенциальным источником прерывания, обращающимся к глобальным данным. Например, прерывание на уровне DPC/dispatch заставляет запускать диспетчер, использующий диспетчерскую базу данных. Поэтому любая другая часть ядра, использующая базу данных диспетчера, поднимает IRQL на уровень DPC/dispatch, маскируя прерывания уровня DPC/dispatch перед использованием базы данных диспетчера.

Эта стратегия хорошо подходит для однопроцессорных систем, но она не отвечает требованиям многопроцессорной конфигурации. Подъем уровня IRQL на одном процессоре не предотвращает возникновения прерывания на другом процессоре. Ядру также нужно гарантировать взаимно исключающий доступ для нескольких процессоров.

Взаимоблокируемые операции

Простейшая форма механизмов синхронизации для безопасной работы с целочисленными значениями и для проведения сравнений в мультипроцессорной среде полагается на аппаратную поддержку. Эти механизмы включают в себя такие функции, как InterlockedIncrement, InterlockedDecrement, InterlockedExchange и InterlockedCompareExchange. К примеру, функция InterlockedDecrement использует используемый в системе x86 префикс инструкций lock (например, lock xadd) для блокировки мультипроцессорной шины при операциях вычитания. Это делается для того, чтобы другой процессор, который также изменяет то же самое место в памяти, подвергающееся уменьшению значения, не мог его изменить в тот период, когда уменьшающий значение процессор читает исходное значение и записывает обратно уменьшенное значение. Эта форма базовой синхронизации используется ядром и драйверами. В современном комплекте компиляторов Microsoft такие функции называются *встроенными* (intrinsic), потому что код для них генерируется встроенным ассемблером непосредственно в фазе компиляции, вместо того чтобы проходить через вызов функции. Вполне вероятно, что помещение параметров в стек, вызов функции, копирование параметров в регистры, а затем извлечение параметров из стека и возвращение управления вызывающему коду будут более затратными операциями, чем та работа, которой функция должна заниматься в первую очередь.

Спин-блокировки

Механизм, используемый ядром для достижения мультипроцессорной взаимной блокировки, называется спин-блокировкой. Спин-блокировка представляет собой блокирующий примитив, связанный с глобальной структурой данных, например с DPC-очередью, показанной на рис. 3.25.

Перед входом в любой критический раздел, показанный на рис. 3.25, ядро должно получить спин-блокировку, связанную с защищенной DPC-очередью. Если спин-блокировка занята, ядро предпринимает попытки ее получения, пока не добьется успеха. Спин-блокировка получила свое название на основе того факта, что ядро (и, таким образом, процессор) ждет, «прокручивая» код (занимаясь спиннингом), пока не получит блокировку.

Спин-блокировки, как и защищаемые ими структуры данных, находятся в невыгружаемой памяти, отображаемой на системное адресное пространство. Код для получения и освобождения спин-блокировки с целью обеспечения приемле-

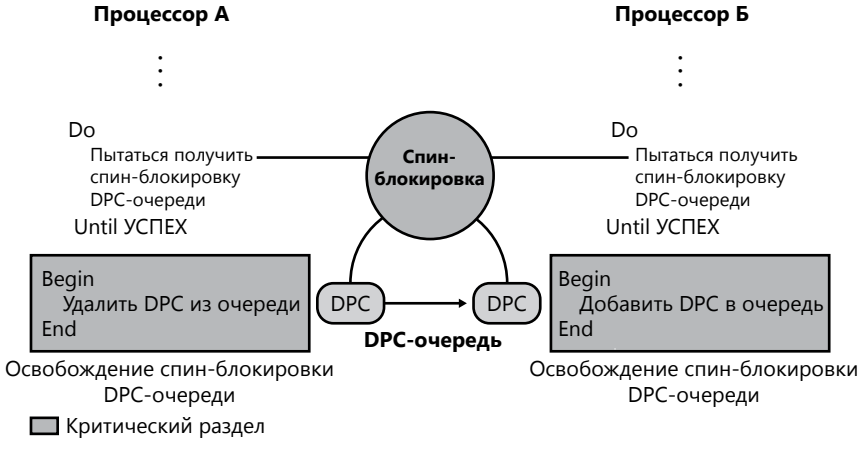


Рис. 3.25. Использование спин-блокировки

мой скорости работы и использования того блокирующего механизма, который предоставляется базовой архитектурой процессора, пишется на языке ассемблера. На многих архитектурах спин-блокировки реализованы с помощью аппаратно поддерживаемой операции «проверки и установки» (test-and-set), которая проверяет значение переменной блокировки и получает блокировку в рамках одной атомарной инструкции. Проверка и получение блокировки являются одной инструкцией, исключающей возможность захвата блокировки вторым потоком в промежуток между тестированием переменной первым потоком и получением им этой блокировки. Кроме того, упомянутая ранее инструкция lock также может быть использована в операции «проверки и установки», в виде комбинированной операции на ассемблере lock bts, которая также блокирует шину мультипроцессора; в противном случае атомарное выполнение операции было бы возможным на более чем одном процессоре. Без инструкции lock операция может быть гарантированно атомарной только на текущем процессоре.

В Windows все спин-блокировки в режиме ядра имеют связанный с ними IRQL, который всегда на уровне DPC/dispatch или выше. Таким образом, когда поток пытается получить спин-блокировку, все остальные действия на IRQL спин-блокировки или ниже этого уровня на этом процессоре прекращаются. Поскольку диспетчеризация потоков происходит на уровне DPC/dispatch, поток, удерживающий спин-блокировку, никогда не прерывается, поскольку IRQL маскирует механизмы диспетчеризации. Эта маскирование позволяет коду, выполняемому в критическом разделе и защищенному спин-блокировкой, продолжить выполнение, чтобы он мог быстро освободить блокировку. В ядре спин-блокировки используются весьма осмотрительно, минимизируя количество инструкций, выполняемых ядром при удержании спин-блокировки. Любой процессор, пытающийся получить спин-блокировку, по сути, будет занят бесконечным ожиданием, потребляя энергию (ожидание в режиме занятости приводит к 100 % использованию центрального процессора), и не будет выполнять никакой полезной работы.

На процессорах x86 и x64 в циклы ожидания в режиме занятости может быть вставлена специальная ассемблерная инструкция pause. Эта инструкция подсказывает процессору, что обрабатываемые им инструкции цикла являются частью

спин-блокировки (или подобной ей конструкции), требующей цикл. Инструкция дает три преимущества:

- Она существенно сокращает потребление энергии за счет небольшой задержки в работе ядра вместо постоянного выполнения цикла.
- На ядрах с технологией HyperThread она позволяет центральному процессору понять, что «работа», выполняемая тем логическим ядром, которое занимается спиннингом, не относится к слишком важным, и выделить больше времени центрального процессора второму логическому ядру.
- Поскольку цикл ожидания в режиме занятости приводит к настоящему взрыву запросов на чтение, выставляемых на шину ожидающими потоками (которые могут генерироваться в неподходящие моменты), центральный процессор пытается исправить нарушения в порядке доступа к памяти, как только он обнаруживает запись (то есть когда владеющий блокировкой поток ее освобождает). Таким образом, как только спин-блокировка будет освобождена, центральный процессор изменит порядок любых отложенных операций чтения из памяти для обеспечения правильного порядка. Это изменение порядка выражается в больших издержках производительности системы и его можно избежать с помощью инструкции pause.

Ядро открывает доступ к спин-блокировкам другим частям исполняющей системы посредством набора функций ядра, в котором есть такие функции, как KeAcquireSpinLock и KeReleaseSpinLock. Например, драйверы устройств запрашивают спин-блокировки, чтобы обеспечить в отдельно взятый момент времени доступ к регистрам устройства и другим глобальным структурам данных только из одной части драйвера устройства (и только с одного процессора). Спин-блокировки не предназначены для использования пользовательскими программами, эти программы должны использовать объекты, рассматриваемые в следующем разделе. Драйверам устройств также нужно защитить доступ к своим собственным структурам данных от связанных с этими же драйверами прерываний. Поскольку API-функции спин-блокировки обычно поднимают IRQL только до уровня DPC/dispatch, этого недостаточно для защиты от прерываний. По этой причине ядро также экспортирует API-функции KeAcquireInterruptSpinLock и KeReleaseInterruptSpinLock, которые берут в качестве параметра объект KINTERRUPT, рассмотренный в начале данной главы. Система ищет в объекте прерывания связанные с прерыванием DIRQL и поднимает IRQL на соответствующий уровень для обеспечения должного доступа к структурам, используемым совместно с ISR. Для синхронизации целой функции с ISR, вместо простого использования критического раздела, устройства могут использовать API-функцию KeSynchronizeExecution. Во всех случаях код, защищенный спин-блокировкой прерывания, должен выполняться очень быстро, поскольку любые задержки станут причиной задержек прерывания, выходящих за пределы обычных норм, и окажут существенное отрицательное влияние на производительность системы.

Спин-блокировки ядра накладывают на использующий их код ряд ограничений. Поскольку спин-блокировки всегда имеют IRQL-уровень равный или более высокий, чем DPC/dispatch, то, как уже ранее объяснялось, код, удерживающий

спин-блокировку, вызовет сбой системы, если попытается заставить планировщика выполнить операцию диспетчеризации или если его выполнение станет причиной ошибки обращения к странице.

Спин-блокировки с очередями

Чтобы улучшить масштабируемость спин-блокировок, во многих случаях вместо стандартной спин-блокировки используется так называемая *спин-блокировка с очередью*. Она работает следующим образом: когда процессор хочет получить спин-блокировку с очередью, которая в этот момент удерживается, он помещает свой идентификатор в очередь, связанную со спин-блокировкой. Когда процессор, удерживающий спин-блокировку, ее освобождает, он передает спин-блокировку первому процессору, идентифицированному в очереди. Тем временем тот процессор, который ждал занятую спин-блокировку, проверяет состояние не самой спин-блокировки, а флага, имеющегося у каждого процессора на предмет того, что процессор, стоящий перед ним в этой очереди, установил этот флаг, чтобы показать, что настала очередь ожидающего процессора.

Тот факт, что спин-блокировки с очередью приводят к спиннингу вокруг флагов, имеющихся у каждого процессора, а не вокруг глобальных спин-блокировок, приводит к двум последствиям. Во-первых, шина мультипроцессора становится не такой загруженной трафиком со стороны межпроцессорной синхронизации. Во-вторых, вместо случайного процессора в группе ожидания получения спин-блокировки, спин-блокировка с очередью устанавливает порядок доступа к блокировке под названием «первым пришел, первым ушел» (FIFO). Такой порядок означает более упорядоченную работу тех процессоров, которые имеют доступ к одним и тем же блокировкам.

Windows определяет несколько глобальных блокировок с очередью, сохраняя указатели на них в массиве, который содержится в *блоке управления областью процессора* (processor region control block, PRCB). Подобный блок имеется в каждом процессоре. Глобальная спин-блокировка может быть получена путем вызова функции KeAcquireQueuedSpinLock с индексом того элемента PRCB-массива, в которм содержится указатель на спин-блокировку. Количество глобальных спин-блокировок вырастает с каждым выпуском операционной системы, а таблица определения индексов для них публикуется в заголовочном файле WDK с именем Wdm.h. Но при этом следует учесть, что получение одной из таких спин-блокировок с очередью от драйвера устройства является неподдерживаемой и крайне нежелательной операцией. Эти блокировки предназначены исключительно для внутреннего использования ядром операционной системы.

ЭКСПЕРИМЕНТ: ПРОСМОТР ГЛОБАЛЬНЫХ СПИН-БЛОКИРОВОК С ОЧЕРЕДЬЮ

Состояние глобальных спин-блокировок с очередью (тех, указатели на которые содержатся в массиве спин-блокировок с очередью PCR-блока каждого процессора) можно просмотреть путем использования команды отладчика ядра !qlocks. В следующем примере спин-блокировка с очередью, относящаяся к базе данных номеров страничных блоков (page frame number, PFN), удерживается процессором 1, а другие спин-блокировки с очередью не получены.

```
lkd> !qlocks
Key: O = Owner, 1-n = Wait order, blank = not owned/waiting, C = Corrupt
                      Processor Number
    Lock Name         0  1
KE   - Unused Spare
MM   - Expansion
MM   - Unused Spare
MM   - System Space
CC   - Vacb
CC   - Master                                                    ■
```

Внутристековые спин-блокировки с очередью

Драйверы устройств могут использовать динамически выделяемые спин-блокировки с очередью с помощью функций KeAcquireInStackQueuedSpinLock и KeReleaseInStackQueuedSpinLock. Ряд компонентов, включая диспетчер кэш-памяти, диспетчер пула исполняющей системы и NTFS, пользуются такими типами блокировки вместо глобальных спин-блокировок с очередью.

Функция KeAcquireInStackQueuedSpinLock берет указатель на структуру данных спин-блокировки и на дескриптор очереди спин-блокировки. Дескриптор спин-блокировки представляет собой структуру данных, в которой ядро хранит информацию о состоянии блокировки, включая информацию о владельце блокировки и об очереди процессоров, которые могут находиться в процессе ожидания доступности блокировки. По этой причине дескриптор не должен быть глобальной переменной. Обычно это переменная стека, обеспечивающая *местоположение* вызывающему потоку, и отвечающая за *InStack*-часть спин-блокировки и имя API-функции.

Взаимоблокируемые операции исполняющей системы

Для более сложных операций, таких как добавление и удаление записей в списках с одинарными и двойными связями, ядро поддерживает ряд простых функций синхронизации, построенных на спин-блокировках. В качестве примера можно привести функции ExInterlockedPopEntryList и ExInterlockedPushEntryList для списков с одинарной связью, и функции ExInterlockedInsertHeadList и ExInterlocked-RemoveHeadList для списков с двойной связью. Все эти функции требуют стандартной спин-блокировки в качестве параметра и используются ядром и драйверами устройств.

Вместо того чтобы для получения и освобождения параметра спин-блокировки полагаться на стандартные API-функции, эти функции встраивают нужный код, а также используют другую схему очередности. Тогда как API-функции спин-блокировки, имеющие префикс Ke, сначала тестируют и устанавливают бит, для того чтобы посмотреть, не освобождена ли блокировка, а затем для фактического получения блокировки проводят ее в рамках атомарной операции по принципу «проверки и установки». Эти процедуры запрещают прерывания процессора и сразу же пытаются провести атомарную операцию «проверки и установки». Если начальная попытка будет неудачной, прерывания снова разрешаются и про-

должается выполнение стандартного алгоритма ожидания в режиме занятости до тех пор, пока операция «проверки и установки» не вернет 0, в случае чего снова перезапускается вся функция. В силу этих тонких отличий, спин-блокировка, используемая для взаимоблокируемых функций, не должна использоваться с ранее рассмотренными стандартными API-функциями ядра. Конечно, операции со списками, не использующие взаимную блокировку, не должны смешиваться со взаимоблокируемыми операциями.

ПРИМЕЧАНИЕ

Некоторые взаимоблокируемые операции исполняющей системы по возможности молча игнорируют спин-блокировку. Например, API-функции ExInterlockedIncrementLong или ExInterlockedCompareExchange фактически используют тот же самый префикс lock, который используется стандартными взаимоблокируемыми функциями и встроенными функциями. Эти функции были полезны на старых системах (или системах, не относящихся к семейству x86), где операция lock была неприемлема или недоступна. Поэтому теперь вместо этих нерекомендуемых вызовов предпочтение отдается встроенным функциям.

Низкоуровневая IRQL-синхронизация

Программное обеспечение исполняющей системы, не входящее в состав ядра, также нуждается в синхронизации доступа к глобальной структуре данных с мультипроцессорной средой. Например, у диспетчера памяти есть только одна база данных страничных блоков, к которой он обращается как к глобальной структуре данных, а драйверам устройств нужно обеспечить возможность получения исключающего доступа к их устройствам. Путем вызова функций ядра исполняющая система может создать спин-блокировку, получить ее и освободить эту блокировку.

Но спин-блокировки только лишь частично удовлетворяют потребности исполняющей системы в механизмах синхронизации. Поскольку ожидание спин-блокировки буквально парализует процессор, спин-блокировки могут использоваться только при следующих, весьма ограниченных обстоятельствах:

- Доступ к защищаемому ресурсу должен осуществляться быстро и без сложных взаимодействий с остальным кодом.
- Критический раздел кода не должен попадать в страницы, выгружаемые из памяти, не должен ссылаться на данные, находящиеся в выгружаемой памяти, не должен вызывать внешние процедуры (включая системные службы) и не должен генерировать прерывания или исключения.

Эти ограничения являются исчерпывающими и не могут соблюдаться при всех возможных обстоятельствах. Кроме того, исполняющей системе нужно выполнять другие виды синхронизации в добавок к взаимному исключению, и она должна также предоставлять механизмы синхронизации пользовательскому режиму.

Когда спин-блокировки не подходят, можно воспользоваться следующими дополнительными механизмами синхронизации:

- объектами диспетчера ядра (kernel dispatcher objects);
- быстрыми мьютексами (fast mutexes) и защищенными мьютексами (guarded mutexes);

- пуш-блокировками;
- ресурсами исполняющей системы.

Кроме того, у кода пользовательского режима, который также выполняется при низком уровне IRQL, должны быть свои собственные примитивы блокировки. Windows поддерживает различные примитивы, присущие пользовательскому режиму:

- переменные условий (CondVars);
- гибкие блокировки чтения-записи — Slim Reader-Writer Locks (SRW Locks);
- однократно запускаемая инициализация — Run-once initialization (InitOnce);
- критические разделы (Critical sections).

Мы рассмотрим примитивы пользовательского режима и лежащую в их основе поддержку со стороны ядра чуть позже, а теперь давайте сосредоточимся на объектах режима ядра. Таблица 3.18 служит справочником, в котором сравниваются и противопоставляются возможности этих механизмов и их взаимодействие с доставкой APC режима ядра.

Таблица 3.18. Механизмы синхронизации режима ядра

	Открытость для использования драйверами устройств	**Отключение обычных APC-вызовов режима ядра**	**Отключение специальных APC-вызовов режима ядра**	**Поддержка рекурсивного получения**	**Поддержка общедоступного и исключительного получения**
Мьютексы диспетчера ядра	Да	Да	Нет	Да	Нет
Семафоры или события диспетчера ядра	Да	Нет	Нет	Нет	Нет
Быстрые мьютексы	Да	Да	Да	Нет	Нет
Защищенные мьютексы	Да	Да	Да	Нет	Нет
Пуш-блокировки	Нет	Нет	Нет	Нет	Да
Ресурсы исполнительной системы	Да	Нет	Нет	Да	Да

Объекты диспетчера ядра

Ядро предоставляет исполняющей системе дополнительные механизмы синхронизации в форме объектов ядра, известных под общим названием *объекты диспетчера*. Видимые API-функциям Windows объекты синхронизации получают свои синхронизационные возможности от этих объектов диспетчера ядра.

Каждый объект, видимый Windows API, который поддерживает синхронизацию, инкапсулирует, как минимум, один объект диспетчера ядра.

Семантика синхронизации исполняющей системы видима программистам Windows programmers посредством функций WaitForSingleObject и WaitForMultipleObjects, которые реализуются в подсистеме Windows путем вызова аналогичных системных служб, предоставляемых диспетчером объектов. Поток в Windows-приложении может синхронизироваться с помощью множества различных объектов, включая процессы, потоки, события Windows, семафоры, мьютексы, таймеры ожидания, порт завершения ввода-вывода, ALPC-порт, раздел реестра или файловый объект. Фактически, ожидание можно организовать почти на всех объектах, показываемых ядром. Одни из них являются настоящими объектами диспетчера, а другие являются более крупными объектами, имеющими внутри себя объекты диспетчера (например, порты, разделы или файлы). В табл. 3.19 показаны настоящие объекты диспетчера, следовательно, все остальные объекты, на которых Windows API позволяется ожидание, наверное, содержат один из таких примитивов.

Еще один тип объекта синхронизации исполняющей системы, о котором стоит упомянуть, называется *ресурсом исполняющей системы*. Ресурсы исполняющей системы обеспечивают исключающий доступ (подобно мьютексу), а также совместный доступ по чтению (несколько читателей совместно используют к структуре доступ по чтению). Но они доступны только коду режима ядра и поэтому недоступны из Windows API. В остальных подразделах рассматриваются подробности реализации ожидания объектов диспетчера.

Ожидание объектов диспетчера

Поток может быть синхронизирован с помощью объекта диспетчера путем ожидания дескриптора объекта. Это заставит ядро перевести поток в режим ожидания.

В любой отдельно взятый момент времени объект синхронизации находится в одном из двух состояний: в *сигнальном состоянии* или в *несигнальном состоянии*. Поток не может возобновить свое выполнение до тех пор, пока не будет удовлетворено его ожидание, а это условие будет выполнено, когда объект диспетчера, дескриптор которого ожидается потоком, также претерпит изменение состояния с несигнального состояния в сигнальное состояние (когда другой поток установит, к примеру, объект события). Для синхронизации с помощью объекта поток вызывает одну из системных служб ожидания, предоставляемую диспетчером объектов, передавая дескриптор объекта, с которым он хочет быть синхронизирован. Поток может ждать один или несколько объектов и может также указать, что его ожидание может быть прервано, если оно не завершилось в течение определенного периода времени. Когда ядро устанавливает объект в сигнальное состояние, одна из имеющихся в ядре сигнальных процедур проверяет, не ожидает ли этот объект какие-либо потоки, которые больше не ожидают перехода в сигнальное состояние никаких других объектов. Если такие потоки имеются, ядро освобождает один или несколько потоков от их ожидающего состояния, позволяя им продолжить выполнение.

Проиллюстрировать порядок взаимодействия синхронизации с диспетчеризацией потоков можно на следующем примере установки события:

1. Поток пользовательского режима ожидает дескриптора объекта события.
2. Ядро изменяет состояние диспетчеризации потока на ожидание, а затем добавляет поток к списку потоков, ожидающих событие.
3. Какой-то другой поток устанавливает событие.
4. Ядро проходит вниз по списку потоков, ожидающих событие. Если удовлетворены условия ожидания потока (см. следующее примечание), ядро выводит поток из состояния ожидания. Если это поток изменяемого приоритета, ядро может также поднять его приоритет выполнения (см. главу 5).

ПРИМЕЧАНИЕ

Некоторые потоки могут ожидать более одного объекта, поэтому они продолжают ожидание, если только ими не указано ожидание любого из объектов — WaitAny, что их разбудит, как только в сигнальное состояние перейдет один объект (а не все объекты).

Что переводит объект в сигнальное состояние?

Сигнальное состояние определяется для разных объектов по-разному. Объект потока имеет несигнальное состояние в течение всего своего жизненного цикла и устанавливается в сигнальное состояние ядром, когда выполнение потока завершается. Подобным же образом ядро устанавливает объект процесса в сигнальное состояние, когда завершается выполнение последнего потока этого процесса. В отличие от этих объектов объект таймера, подобно будильнику, устанавливается в положение «состоявшегося» в определенное время. Когда его время истекает, ядро устанавливает объект таймера в сигнальное состояние.

При выборе механизма синхронизации в программе нужно учитывать правила управления поведением различных объектов синхронизации. Порядок завершения ожидания потока при переводе объекта в сигнальное состояние зависит от типа того объекта, который ожидается потоком, что проиллюстрировано в табл. 3.19.

Таблица 3.19. Определения сигнального состояния

Тип объекта	Устанавливается в сигнальное состояние, когда	Влияние на ожидающие потоки
Процесс	Завершается последний поток	Освобождаются все потоки
Поток	Завершается поток	Освобождаются все потоки
Событие (уведомительного типа)	Поток устанавливает событие	Освобождаются все потоки
Событие (синхронизирующего типа)	Поток устанавливает событие	Один поток освобождается и может получить повышение приоритета; объект события сбрасывается
Шлюз (блокирующего типа)	Поток сообщает о шлюзе	Освобождается и получает более высокий приоритет первый ожидающий поток

Тип объекта	Устанавливается в сигнальное состояние, когда	Влияние на ожидающие потоки
Шлюз (сигнализирующего типа)	Поток сообщает о типе	Освобождается первый ожидающий поток
Событие, снабженное ключом	Поток устанавливает событие с помощью ключа	Освобождается поток, ожидающий ключа и относящийся к тому же процессу, что и сигнализирующий поток
Семафор	Счетчик семафора уменьшается на 1	Освобождается один поток
Таймер (уведомительного типа)	Настало установленное время, или истек интервал времени	Освобождаются все потоки
Таймер (синхронизирующего типа)	Настало установленное время, или истек интервал времени	Освобождается один поток
Мьютекс	Поток освобождает мьютекс	Освобождается один поток, который становится владельцем мьютекса
Очередь	Элемент помещается в очередь	Освобождается один поток

Когда объект устанавливается в сигнальное состояние, ожидающие потоки, в основном, тут же освобождаются от своих ожидающих состояний. Некоторые объекты диспетчера ядра и системные события, которые приводят к изменению их состояния, показаны на рис. 3.26.

Например, объект уведомительного события (называемый в Windows API событием с ручным сбросом) используется для объявления о наступлении какого-нибудь события. Когда объект события переводится в сигнальное состояние, освобождаются все потоки, ожидавшие это событие. Исключение составляет любой поток, одновременно ожидающий более одного объекта, такому потоку может понадобиться продолжить ожидание, пока еще один объект не достигнет сигнального состояния.

В отличие от объекта события объект мьютекса имеет связанного с ним владельца (если только он не был приобретен во время DPC-вызова). Он используется для получения взаимно исключающего доступа к ресурсу, и только один поток в одно и то же время может удерживать мьютекс. Когда объект мьютекса становится свободным, ядро устанавливает его в сигнальное состояние, а затем выбирает для выполнения, а также для наследования любого примененного повышения приоритета один из ожидающих потоков (см. главу 5). Поток, выбранный ядром, получает объект мьютекса, а все остальные потоки продолжают ожидание.

Объект мьютекса может быть также заброшен: это случается, когда завершается выполнение того потока, который им обладает в данный момент. Когда завершается выполнение потока, ядро определяет общее количество мьютексов, которыми владел поток, и переводит их в заброшенное состояние, которое,

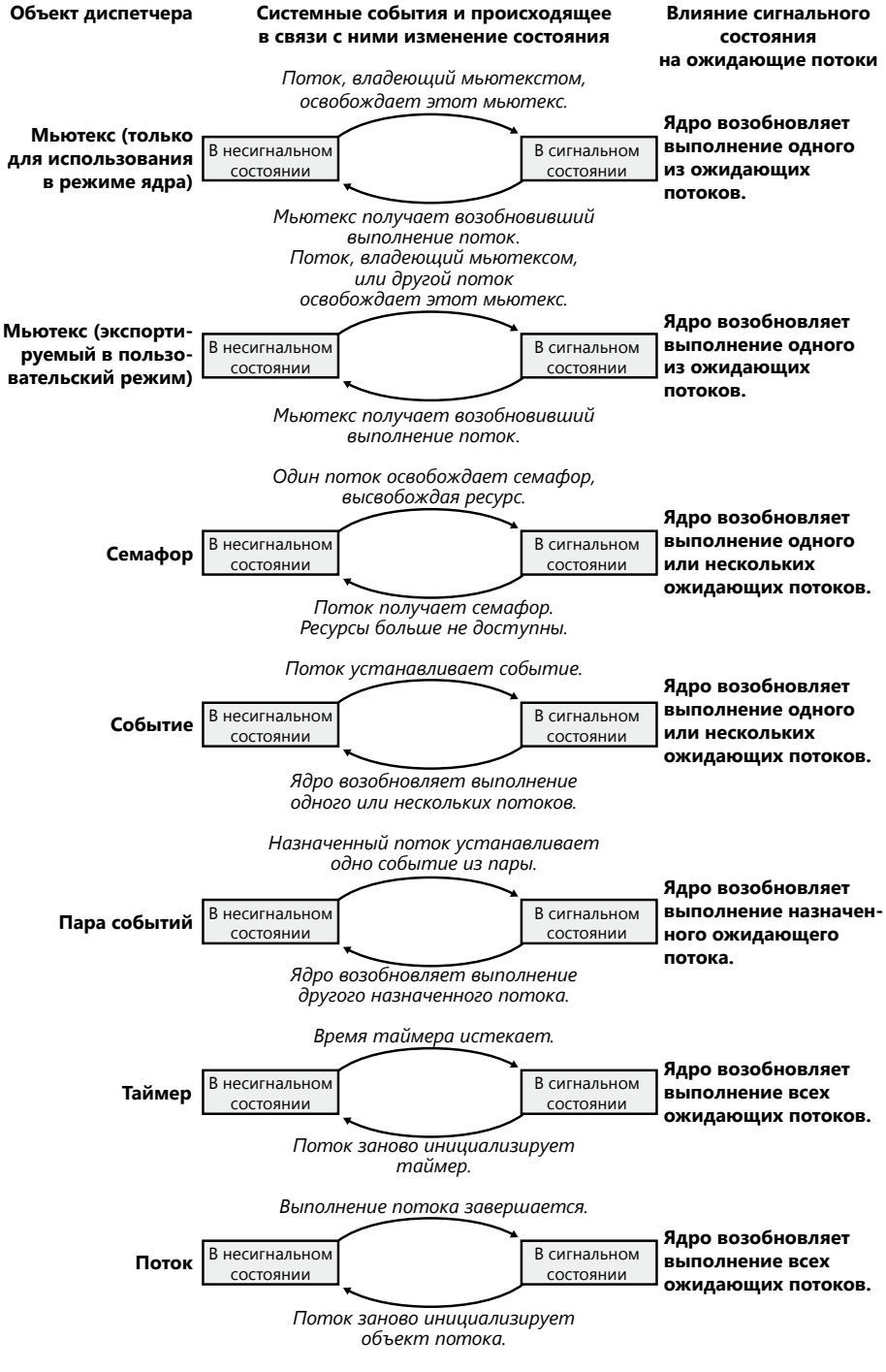


Рис. 3.26. Избранные объекты диспетчера ядра

в понятиях логики сигнализации, рассматривается как сигнальное состояние, в котором владение мьютексом передается ожидающему потоку.

Этот краткий обзор не предназначался для перечисления всех причин и особенностей использования различных объектов исполняющей системы. В нем перечислены их основные функциональные возможности и поведение, связанное с синхронизацией. Для получения информации о том, как воспользоваться этими объектами в программах Windows, обратитесь к справочной документации Windows по объектам синхронизации или к книге авторов Джеффри Рихтера и Кристофера Назара «Windows via C/C++. Программирование на языке Visual C++».

Структуры данных

Для отслеживания, *кто* находится в состоянии ожидания, *как* осуществляется это ожидание, *что именно* ожидается и в *каком состоянии* находится вся операция ожидания, есть три ключевые структуры данных. Эти три структуры — заголовок диспетчера, блок ожидания и регистр состояния ожидания. Первые две структуры открыто определены в файле Wdm.h, включаемом в WDK, а последняя структура не документирована.

Заголовок диспетчера является упакованной структурой, поскольку в нем нужно хранить большой объем информации в структуре фиксированного размера. Одной из основных тонкостей является определение взаимоисключающих флагов в структуре, в одном и том же месте памяти (с одинаковым смещением). С помощью поля Type, ядро знает, какое из этих полей применяется в данном случае. Например, мьютекс может быть заброшенным, а таймер может быть абсолютным или относительным. Аналогичным образом, таймер может быть вставлен в список таймеров, а поле активности отладчика Debug Active имеет смысл только для процессов. С другой стороны, заголовок диспетчера действительно содержит информацию, общую для любого объекта диспетчера: тип объекта, сигнальное состояние и список потоков, ожидающих этот объект.

Блок ожидания представляет поток, ожидающий объект. У каждого потока, находящегося в состоянии ожидания, есть список блоков ожидания, который дает представление об объектах, ожидаемых потоком. У каждого объекта диспетчера имеется список блоков ожидания, который дает представление о том, какие потоки ожидают объект. Этот список содержится так, чтобы при переходе объекта диспетчера в сигнальное состояние ядро могло быстро определить, кто ждет этот объект. И наконец, поскольку поток диспетчера установки баланса (см. главу 5), запускаемый на каждом центральном процессоре, нуждается в анализе времени пребывания каждого потока в режиме ожидания (с целью принятия решения о том, нужно ли выгружать страницу стека ядра), у каждого PRCB есть список ожидающих потоков.

Блок ожидания содержит указатель на ожидаемый объект, указатель на поток, ожидающий объект, и указатель на следующий блок ожидания (если поток ожидает более одного объекта). В нем также записывается тип ожидания (какой-нибудь или все), а также позиция элемента массива дескрипторов, переданная потоком при вызове функции WaitForMultipleObjects (позиция 0, если поток ожидал только один объект). Тип ожидания имеет очень важное значение при

удовлетворении ожидания, поскольку он определяет, принадлежат или нет все блоки ожидания тому процессу, который ожидает сигнальный объект, находящийся в обработке: при ожидании любого из объектов диспетчер не интересуется состоянием других объектов, поскольку сигнал поступил хотя бы от одного (текущего) объекта. В другой стороны, при ожидании всех объектов диспетчер должен возобновлять выполнение потока только тогда, когда все остальные объекты также находятся в сигнальном состоянии, чтобы узнать это, нужно обойти все блоки ожидания и связанные с ними объекты.

Блок ожидания также содержит изменяющийся показатель своего состояния, который определяет текущее состояние этого блока ожидания в транзакционной операции ожидания, которой он занимается в данный момент. Различные состояния, их значения и производимые эффекты рассматриваются в табл. 3.20.

Таблица 3.20. Состояния блока ожидания

Состояние	Значение	Эффект
WaitBlockActive (2) (блок ожидания активен)	Данный блок ожидания активно связан с объектом как с частью потока, который находится в режиме ожидания	Во время удовлетворения ожидания этот блок ожидания будет отсоединен от списка блоков ожидания
WaitBlock-Inactive (3) (блок ожидания неактивен)	Поток, связанный с данным блоком, был удовлетворен (или срок, если таковой на это отводится, уже истек)	Во время удовлетворения ожидания этот блок ожидания не будет отсоединен от списка блоков ожидания, потому что удовлетворение ожидания уже должно было отсоединить его от списка, когда этот блок находился в активном состоянии
WaitBlock-BypassStart (0) (начался обход блока ожидания)	Сигнал доставляется потоку, но он еще не перешел в режим ожидания	Во время удовлетворения (которое произошло бы немедленно, еще до входа потока в настоящее состояние ожидания) ожидающий поток должен синхронизироваться с сигнальным объектом, поскольку существует риск того, что объект ожидания может быть в стеке — пометка блока неактивным заставит ожидающий поток извлечь данные из стека, в то время как сигнальный объект может быть все еще ему доступен
WaitBlock-BypassComplete (1) (обход блока ожидания завершен)	Теперь ожидающий поток, связанный с этим блоком ожидания, правильно синхронизирован (удовлетворение ожидания завершено), и сценарий обхода завершен	Теперь блок ожидания рассматривается, по сути, как неактивный (игнорируемый) блок ожидания

Общее состояние потока (или любого из объектов, требуемого для начала ожидания) может измениться в то время, пока операции ожидания все еще подготавливаются. Ничто не мешает другому потоку, выполняемому на другом

логическом процессоре, попытаться ввести в сигнальное состояние объекты, или, может быть, оповестить поток, или даже отправить ему APC-вызов. Диспетчер ядра нуждается в отслеживании двух дополнительных фрагментов данных для каждого ожидающего потока: текущее четко определяемое состояние ожидания потока, а также любые изменения подвешенного состояния, способные изменить результат попытки провести операцию ожидания.

Когда потоку дается инструкция ожидания заданного объекта (например, через вызов функции WaitForSingleObject), он сначала пытается войти в действующее состояние ожидания (WaitInProgress), приступая к ожиданию. Эта операция достигает успеха, если на данный момент времени в адрес потока нет отложенных оповещений. Здесь все основано на способности получения оповещений в режиме ожидания и текущем режиме ожидания процессора, которые определяют, может ли оповещение получить приоритет над ожиданием. Если оповещение имеется, вход в режим ожидания вообще не происходит, и вызывающая программа получает соответствующий код состояния. В противном случае поток входит в состояние WaitInProgress, и в этот момент основное состояние потока устанавливается в Waiting (ожидающий). При этом записываются причина и время ожидания, кроме того, должны быть зарегистрированы какие-нибудь сроки истечения ожидания.

После входа в режим ожидания поток может нужным образом инициализировать блоки ожидания (и, в процессе, пометить их как WaitBlockActive), а затем приступить к отслеживанию всех объектов, являющихся частями этого ожидания. Поскольку у каждого объекта есть своя собственная блокировка, важно, чтобы ядро могло обеспечить последовательную схему очередности блокировки, при которой несколько процессоров могли бы анализировать цепочку ожиданий, состоящую из многих объектов (выстроенную с помощью вызова функции WaitForMultipleObjects). Для этого ядром используется технология, известная как *выстраивание порядка адресов* (address ordering). Поскольку у каждого объекта есть отличный от других и статичный адрес режима ядра, объекты могут быть выстроены в монотонно-возрастающем порядке адресов, гарантирующем, что блокировки будут всегда получаться и освобождаться в одном и том же порядке всеми вызывающими программами. Это означает, что соответствующим образом может быть продублирован и отсортирован массив объектов, предоставляемый вызывающей программой.

Следующим этапом будет проверка возможности немедленного удовлетворения ожидания, например, когда потоку предписывается ожидание мьютекса, который уже освободился, или события, о наступлении которого уже был получен сигнал. В таких случаях ожидание удовлетворяется немедленно, что включает в себя отсоединение связанных с этим блоков ожидания (в данном случае никакие блоки ожидания еще и не были вставлены) и выполнение выхода из ожидания (обработка любых, отложенных операций планировщика, помеченных в регистре состояния ожидания). Если этот кратчайший путь пройти не удается, ядро пытается проверить, не истекло ли указанное для ожидания время (если таковое имелось). В этом случае ожидание не «удовлетворяется», а просто считается «просроченным», что приводит к немного более быстрой обработке кода выхода, хотя и с таким же результатом.

Если ни один из этих кратчайших путей не был пройден, блок ожидания вставляется в список ожидания процесса, и теперь поток пытается совершить свое ожидание. Тем временем блокировка или блокировки объекта освобождается, позволяя другим процессорам изменить состояние любого из объектов, на котором, как теперь предполагается, поток пытается построить свое ожидание. Если предположить развитие событий по сценарию, в котором отсутствуют моменты соперничества, где другие процессоры не интересуются этим потоком или его объектами ожидания, ожидание переключается в состояние совершения ожидания, если нет незавершенных изменений, отмеченных в регистре состояния ожидания. Операция совершения ожидания связывает ожидающий поток со списком PRCB, активирует, если это нужно, дополнительный поток очереди ожидания, и вставляет таймер, связанный с подсчетом времени истечения ожидания, если таковое имеется. Поскольку потенциально к этому моменту завершится довольно много циклов, появится новая возможность истечения лимита времени. При таком сценарии вставка таймера приведет к немедленной отправке сигнала потоку, и, таким образом, к удовлетворению ожидания по таймеру, и к общему истечению времени ожидания. В противном случае, по наиболее распространенному развитию сценария, теперь контекст центрального процессора переключится на следующий поток, готовый к выполнению (см. главу 5).

В условиях высокого уровня соперничества путей выполнения кода, создающихся на мультипроцессорных машинах, есть довольно высокая степень вероятности того, что поток, пытающийся перейти в режим ожидания, уже претерпел изменение, в то время как вопрос ожидания еще решался. Один из возможных сценариев заключается в том, что один из объектов, который ожидался потоком, только что перешел в сигнальное состояние. Как уже ранее говорилось, этот приводит к тому, что связанный с ним блок ожидания входит в состояние WaitBlockBypassStart, и регистр состояния ожидания потока теперь показывает состояние ожидания WaitAborted (ожидание прекращено). Другой возможный сценарий заключается в том, что ожидающему потоку было выдано оповещение или APC-вызов, что не приведет к установке состояния WaitAborted, но установит один из соответствующих разрядов в регистре состояния ожидания. Поскольку APC-вызовы могут прекращать ожидание в зависимости от типа APC-вызова, режима ожидания и возможности принимать оповещения, APC-вызов доставляется, и ожидание прекращается. Другие операции будут изменять регистр состояния ожидания без генерирования полного цикла прекращения, включая изменения, вносимые в состояния приоритетности или родственности потока, которые будут обработаны при выходе из режима ожидания по причине ошибки перехода в этот режим, как уже упоминалось в предыдущих примерах.

На рис. 3.27 показана связь объектов диспетчера с блоками ожидания, с потоками и с PRCB. В данном примере у центрального процессора 0 есть два ожидающих потока (приступивших к ожиданию): поток 1 ожидает объект Б, а поток 2 ожидает объекты А и Б. Если объект А переходит в сигнальное состояние, ядро видит, что, поскольку поток 2 также ожидает еще один объект, этот поток не может быть готов к выполнению. С другой стороны, если в сигнальное состояние переходит объект Б, ядро может сразу же подготовить к выполнению поток 1, потому что он не ждет какие-либо другие объекты. Если бы поток 1 также ожидал другие объекты, но тип его ожидания был бы WaitAny, ядро все равно могло бы возобновить его выполнение.

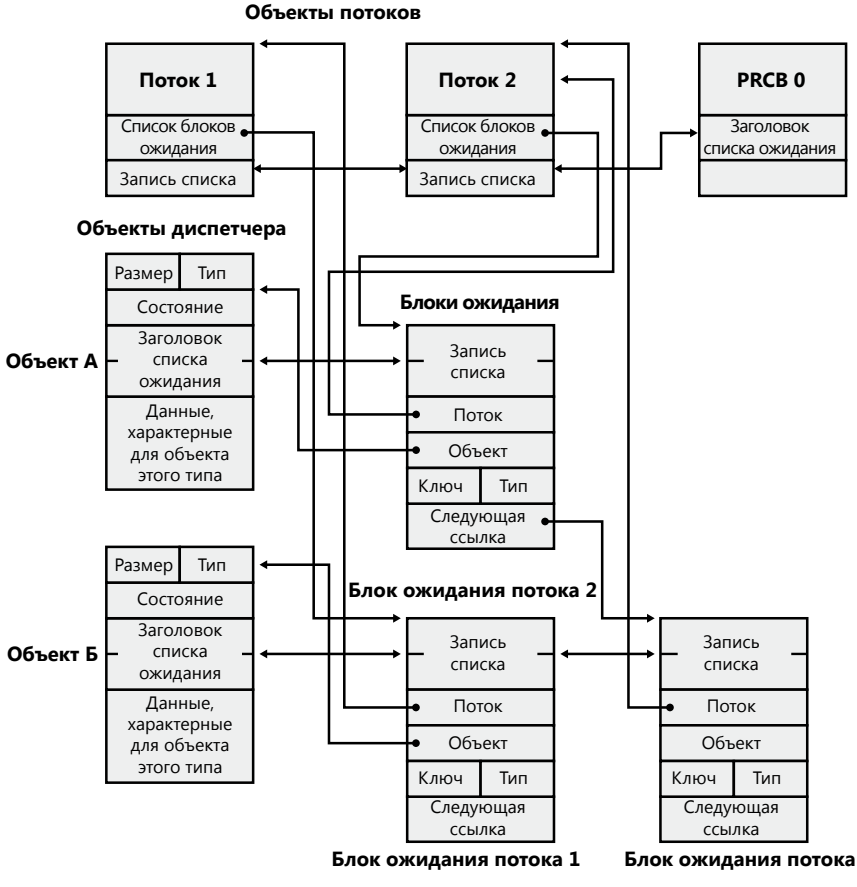


Рис. 3.27. Структуры данных, необходимые для обеспечения ожидания

ЭКСПЕРИМЕНТ: ПРОСМОТР ОЧЕРЕДЕЙ ОЖИДАНИЙ

Список объектов, ожидаемых потоком, можно просмотреть с помощью команды отладчика ядра !thread. Например, следующая выборка из вывода на экран, полученного с помощью команды !process, показывает, что поток ждет объект события:

```
kd> !process
§
   THREAD fffffa8005292060  Cid 062c062c.0660  Teb: 000007fffffde000 Win32Thread:
fffff900c01c68f0 WAIT: (WrUserRequest) UserMode Non-Alertable
       fffffa80047b8240  SynchronizationEvent
```

Для того чтобы разобраться с заголовком диспетчера объекта, можно воспользоваться командой dt:

```
lkd> dt nt!_DISPATCHER_HEADER fffffa80047b8240
   +0x000 Type             : 0x1 ''
   +0x001 TimerControlFlags : 0 ''
   +0x001 Absolute         : 0y0
```

продолжение ⇩

```
+0x001Coalescable            : 0y0
+0x001 KeepShifting          : 0y0
+0x001 EncodedTolerableDelay : 0y00000 (0)
+0x001 Abandoned             : 0 ''
+0x001 Signalling            : 0 ''
+0x002 ThreadControlFlags    : 0x6 ''
+0x002 CpuThrottled          : 0y0
+0x002 CycleProfiling        : 0y1
+0x002 CounterProfiling      : 0y1
+0x002 Reserved              : 0y00000 (0)
+0x002 Hand                  : 0x6 ''
+0x002 Size                  : 0x6
+0x003 TimerMiscFlags        : 0 ''
+0x003 Index                 : 0y000000 (0)
+0x003 Inserted              : 0y0
+0x003 Expired               : 0y0
+0x003 DebugActive           : 0 ''
+0x003 ActiveDR7             : 0y0
+0x003 Instrumented          : 0y0
+0x003 Reserved2             : 0y0000
+0x003 UmsScheduled          : 0y0
+0x003 UmsPrimary            : 0y0
+0x003 DpcActive             : 0 ''
+0x000 Lock                  : 393217
+0x004 SignalState           : 0
+0x008 WaitListHead     : _LIST_ENTRY [ 0xfffffa80'047b8248 - 0xfffffa80'047b8248 ]
```

Нужно проигнорировать все значения, которые не относятся к объекту данного типа, поскольку они могут быть либо неправильно декодированы отладчиком (из-за использования неверного типа или поля), либо просто содержат утратившие силу или неверные данные из предыдущего размещенного значения. Нет никакого определенного соотношения, которое можно было бы увидеть и сделать вывод о том, какое поле к какому объекту применяется, остается только просматривать исходный код ядра Windows или комментарии заголовочных файлов WDK. В табл. 3.21 перечислены флаги заголовка диспетчера и объекты, к которым они применяются.

Таблица 3.21. Использование и предназначение флагов заголовка диспетчера

Флаг	Область применения	Предназначение
Absolute	Таймеры	Указывает на то, что истечение времени является абсолютным, а не относительным показателем
Coalescable	Периодические таймеры	Показывает, может ли к этому таймеру применяться объединение
KeepShiftig	Таймеры, допускающие объединение	Показывает, должен ли диспетчер ядра продолжить попытки сдвига времени истечения срока действия таймера. Когда подгонка добирается до машинного периодического интервала, это флаг в итоге приобретает значение FALSE

Флаг	Область применения	Предназначение
Encoded-ToleableDelay	Таймеры, допускающие объединение	Максимальное значение допустимого сдвига (сдвиг на величину, кратную степени двойки), которое может поддерживать таймер, когда запускается за пределами своей ожидаемой периодичности
Abandoned	Мьютексы	Выполнение потока, удерживающего мьютекс, было завершено
Signaling	Шлюзы	При переходе шлюза в сигнальное состояние нужно повысить уровень приоритета рабочего потока
CpuThrottled	Потоки	Для этого потока включен троттлинг (пропуск тактов) центрального процессора (CPU throttling), как и при запуске в DFSS-режиме (распределенного планировщика справедливого раздела — Distributed Fair-Share Scheduler)
CycleProfiling	Потоки	Для этого потока включен режим профилирования циклов центрального процессора
Counter-Profilng	Потоки	Для этого потока включен режим отслеживания счетчика производительности центрального процессора и профилирования
Size	Все объекты	Размер объекта делится на 4, чтобы поместиться в сигнальный байт
Hand	Таймеры	Индекс в таблице дескрипторов таймеров
Index	Таймеры	Индекс в таблице истечения времени таймеров
Insered	Таймеры	Устанавливается, если таймер был вставлен в таблицу дескрипторов таймеров
Expired	Таймеры	Устанавливается, если время таймера уже истекло
DebugActive	Процессы	Указывает на то, что процесс подвергается отладке
ActiveDR7	Поток	Используются аппаратные контрольные точки, поэтому регистр DR7 активен и должен быть очищен при выполнении операций переключения контекста
Instrumented	Поток	Указывает на то, есть ли у потока аппаратный обратный вызов (поддерживается только на Windows для процессоров x64)
UmsScheduled	Поток	Этот поток относится к рабочему (запланированному) потоку планирования в пользовательском режиме (UMS Worker)
UmsPrimary	Поток	Этот поток является потоком (первичным) UMS-планировщика
DpcActive	Мьютексы	Мьютекс был получен во время DPC-вызова
Lock	Все объекты	Используется для блокировки объекта во время операций ожидания, что необходимо для изменения его состояния или связывания; обычно соответствует биту 7 (0x80) поля Type

Кроме этих флагов в поле Type содержится идентификатор объекта. Этот идентификатор соответствует номеру в перечислении KOBJECTS, дамп которого можно получить с помощью отладчика:

```
lkd> dt nt!_KOBJECTS
   EventNotificationObject = 0
   EventSynchronizationObject = 1
   MutantObject = 2
   ProcessObject = 3
   QueueObject = 4
   SemaphoreObject = 5
   ThreadObject = 6
   GateObject = 7
   TimerNotificationObject = 8
   TimerSynchronizationObject = 9
   Spare2Object = 10
   Spare3Object = 11
   Spare4Object = 12
   Spare5Object = 13
   Spare6Object = 14
   Spare7Object = 15
   Spare8Object = 16
   Spare9Object = 17
   ApcObject = 18
   DpcObject = 19
   DeviceQueueObject = 20
   EventPairObject = 21
   InterruptObject = 22
   ProfileObject = 23
   ThreadedDpcObject = 24
   MaximumKernelObject = 25
```

Если указатели на заголовок списка ожиданий идентичны, то либо объект не ожидает ни один поток, либо его ожидает один поток. При выводе из потока дампа блока ожидания для объекта, являющегося частью нескольких ожиданий, или для объекта, которого ожидают несколько потоков, может получиться следующая картина:

```
dt nt!_KWAIT_BLOCK 0fffffa80'053cf628
   +0x000 WaitListEntry    : _LIST_ENTRY [ 0xfffffa80'02efe568 - 0xfffffa80'02803468 ]
   +0x010 Thread           : 0xfffffa80'053cf520 _KTHREAD
   +0x018 Object           : 0xfffffa80'02803460
   +0x020 NextWaitBlock    : 0xfffffa80'053cf628 _KWAIT_BLOCK
   +0x028 WaitKey          : 0
   +0x02a WaitType         : 0x1 ''
   +0x02b BlockState       : 0x2 ''
   +0x02c SpareLong        : 8
```

Если в списке ожидания более одной записи, можно выполнить ту же самую команду для значения второго указателя в поле WaitListEntry каждого блока ожидания (путем выполнения команды !thread в отношении указателя потока

в блоке ожидания), чтобы пройти по списку и посмотреть, какие еще потоки ожидают объект. Это послужило бы признаком того, что данный объект ожидается несколькими потоками. С другой стороны, при работе с объектом, являющимся частью коллекции объектов, ожидаемой одним потоком, придется вместо этого проанализировать значение поля NextWaitBlock. ■

События с ключом

Объект синхронизации, называемый *событием с ключом*, стоит особого упоминания из-за той роли, которую он играет в примитивах исключающей синхронизации пользовательского режима. События с ключом изначально были реализованы, чтобы помочь процессам справляться с ситуацией дефицита памяти при использовании критических разделов, и они представляют собой объекты синхронизации пользовательского режима. Событие с ключом, которое не задокументировано, позволяет потоку указать «ключ», который он ожидает, и поток возобновляет выполнение, когда другой поток того же процесса сигнализирует о событии с таким же ключом.

Если возникает соперничество, функция EnterCriticalSection динамически размещает объект события, и поток, ожидающий получения критического раздела, ждет, пока тот поток, который владеет критическим разделом, не выдаст ему сигнал, вызвав функцию LeaveCriticalSection. К сожалению, здесь возникает новая проблема. Без событий с ключом система может в критический момент исчерпать объемы памяти, и получение критического раздела может не состояться, поскольку система будет не в состоянии разместить требуемый объект события. Сам по себе дефицит памяти может быть вызван приложением, пытающимся получить критический раздел, поэтому в такой ситуации система войдет в режим взаимной блокировки. Дефицит памяти не единственный сценарий, который может привести к подобному отказу: менее вероятным сценарием может стать дефицит дескрипторов. Если процесс достиг своего 16-миллионного лимита дескрипторов, создание нового дескриптора для объекта события может не состояться.

Отказ, вызванный дефицитом памяти, обычно является исключением из кода, отвечающего за получение критического раздела. К сожалению, в результате повреждается также и критический раздел, что затрудняет отладку и делает объект бесполезным для попытки повторного получения критического раздела. Попытка запуска функции выхода из критического раздела — LeaveCriticalSection, приводит к еще одной попытке размещения объекта события и последующим выдачам исключений и разрушению структуры.

Выделение глобального стандартного объекта события не решит проблему, потому что стандартные примитивы события могут использоваться только для одного объекта. Каждый критический раздел в процессе требует все же своего собственного объекта события, поэтому снова возникнет та же самая проблема. Реализация событий с ключом позволяет нескольким критическим разделам (получение которых ожидается) использовать один и тот же глобальный (для каждого процесса) дескриптор события с ключом. Это позволит критическому разделу работать правильно, даже когда испытывается временный дефицит памяти.

Когда поток посылает сигнал о событии с ключом или ждет этого события, он использует уникальный идентификатор, называемый *ключом*, который идентифицирует экземпляр события с ключом (ассоциируя событие с ключом с отдельным критическим разделом). Когда поток-владелец освобождает событие с ключом,

переводя его в сигнальное состояние, возобновляется выполнение только одного потока, ожидающего этот ключ[1]. Кроме того, возобновляется выполнение только тех ожидающих потоков, которые принадлежат текущему процессу, поэтому ключ одного процесса даже изолирован от ключей других процессов, следовательно, фактически, для всей системы есть только один объект события с ключом. Когда критическим разделом используется событие с ключом, функция входа в критический раздел EnterCriticalSection устанавливает ключ в виде адреса критического раздела и реализует режим ожидания.

Теперь, когда функция EnterCriticalSection вызывает функцию NtWaitForKeyedEvent для реализации ожидания наступления события с ключом, она может дать в качестве параметра для события с ключом дескриптор NULL, сообщая ядру, что создать событие с ключом не представилось возможным. Ядро распознает такое поведение и использует глобальное событие с ключом ExpCritSecOutOfMemoryEvent. Основным преимуществом является то, что процессам не нужно больше тратить объект и его ссылки.

Но события с ключом служат не только альтернативными объектами, применяемыми при дефиците памяти. Когда один и тот же ключ ожидается сразу несколькими потоками, которые нуждаются в возобновлении своего выполнения, о ключе, фактически, сообщается многократно, что требует от объекта вести список всех ожидающих потоков, чтобы в отношении каждого из них можно было провести операцию «пробуждения»[2]. Но поток может отправить сигнал о событии с ключом и без потоков в списке ожиданий. При таком сценарии выдающий сигнал поток сам ждет этого события. Без такой альтернативы поток, выдающий сигнал, может просигналить о событии с ключом в то самое время, когда код пользовательского режима увидел событие в несигнальном состоянии и попытался перейти в режим ожидания. Ожидание, возможно, наступило *после* того, как сигнализирующий поток ввел событие с ключом в сигнальное состояние, в результате чего пропустил импульс, следовательно, ожидающий поток попадает в тупиковую ситуацию. Если заставить сигнализирующий поток ждать по этому сценарию, он, фактически, выдаст сигнал о событии с ключом только тогда, когда кто-то ищет этот сигнал (ждет его).

ПРИМЕЧАНИЕ

Когда сам код ожидания события с ключом нуждается в ожидании, он использует встроенный семафор, который находится в объекте потока режима ядра (ETHREAD), вызывая функцию KeyedWaitSemaphore. (Этот семафор фактически делит свое местоположение с семафором ожидания ALPC.) Дополнительные сведения об объектах потока можно найти в главе 5.

Но события с ключом не заменяют стандартные объекты событий в реализации критических разделов. Изначально причина состояла в том, что во времена Windows XP события с ключом не предлагали масштабируемой производительности в часто используемых сценариях. Следует напомнить, что все описанные алгоритмы предназначались для использования только в критических сценариях, связанных с дефицитом памяти, где важны не только производительность

[1] Такое поведение присуще событиям синхронизации, в отличие от уведомительных событий.

[2] Вспомним, что выдача сигнала о событии с ключом приводит к тому же, что и выдача сигнала о событии синхронизации.

и масштабируемость. Замена стандартного объекта события возложит на события с ключом такую нагрузку, для обработки которой они не предназначались. Основным узким местом в вопросах производительности было то, что события с ключом вели список ожидающих потоков, который реализовывался в виде списка с двойной связью. Списки такого рода имеют низкую *скорость прохода по элементам*, то есть на перебор элементов списка требуется много времени. В данном случае это время зависит от количества ожидающих потоков. Поскольку объект является глобальным, в списке могут быть десятки потоков, требуя большого времени прохода при каждой установке ключа или ожидании события с ключом.

ПРИМЕЧАНИЕ

Начало списка хранится в объекте события с ключом, а потоки связаны через поле KeyedWaitChain (которое совместно используется со временем выхода потока, сохраненного в формате LARGE_INTEGER, имеющем тот же размер, что и список с двойной связью) в объекте потока режима ядра (ETHREAD). Дополнительные сведения об этом объекте даны в главе 5.

Windows повышает производительность событий с ключом за счет использования для хранения ожидающих потоков не связанного списка, а хэш-таблицы. Эта оптимизация позволяет Windows включить в свой состав три новых облегченных примитивов синхронизации пользовательского режима (которые вскоре будут рассмотрены), каждый из которых зависит от события с ключом. Но критические разделы по-прежнему продолжают использовать объекты событий, преимущественно для совместимости приложений и отладки, поскольку объект события и внутренние модули хорошо известны и документированы, а события с ключом непрозрачны и не выставляются для Win32 API.

Быстрые мьютексы и защищенные мьютексы

Быстрые мьютексы, которые так же известны, как *мьютексы исполняющей системы*, обычно предлагают более высокую производительность, чем мьютексные объекты, поскольку, хотя они построены на объектах событий диспетчера, они реализуют ожидание через диспетчер, только если вступают в состязание, в отличие от стандартного мьютекса, попытки приобретения которого всегда ведутся через диспетчер. Это придает быстрым мьютексам особенно высокую производительность в мультипроцессорной среде. Быстрые мьютексы широко используются в драйверах устройств.

Но быстрые мьютексы пригодны к использованию только если может быть отключена доставка обычного APC-вызова режима ядра (рассмотренного ранее в данной главе). В исполняющей системе определяются две функции для их получения: ExAcquireFastMutex и ExAcquireFastMutexUnsafe. Первая из этих функций блокирует все доставки APC путем повышения IRQL-уровня процессора до уровня APC. А вторая предназначена для вызова при обычном отключении APC-доставки в режиме ядра, которое может быть сделано путем повышения IRQL до APC-уровня. Функция ExTryToAcquireFastMutex выполняется аналогично первой из этих двух функций, но она фактически не ждет, если быстрый мьютекс уже удерживается, возвращая вместо этого значение FALSE. Другим ограничением быстрых мьютексов является то, что они не могут быть получены рекурсивно, как это может быть сделано с мьютексными объектами.

Защищенные мьютексы по сути то же самое, что и быстрые мьютексы (хотя в них внутренне используется другой объект синхронизации — KGATE). Их получают с помощью функций KeAcquireGuardedMutex и KeAcquireGuardedMutexUnsafe, но вместо выключения APC-вызовов с помощью повышения IRQL на APC-уровень, они выключают всю APC-доставку режима ядра путем вызова функции KeEnterGuardedRegion. По аналогии с использованием быстрых мьютексов, существует также и метод KeTryToAcquireGuardedMutex. Следует помнить, что в защищенной области, в отличие от критической области, отключаются как специальные, так и обычные APC-вызовы режима ядра, что позволяет защищенному мьютексу избежать повышения IRQL-уровня.

Защищенные мьютексы работают быстрее быстрых мьютексов благодаря трем отличиям:

- Избегая повышения IRQL-уровня, ядро может избежать общения на шине с локальными APIC каждого процессора, являющегося важной операцией на больших SMP-системах. На однопроцессорных системах такой проблемы нет благодаря ленивой оценке IRQL, но для понижения IRQL-уровня может все-таки потребоваться обращение к PIC.
- Оптимизированной версией события является шлюзовый примитив. За счет отсутствия как синхронизирующей, так и уведомительной версии и за счет того, что это исключающий объект, который может ожидаться потоком, код для получения и освобождения шлюза сильно оптимизирован. У шлюзов даже есть вместо получения всей диспетчерской базы данных своя собственная блокировка диспетчера.
- В отсутствие конкуренции при получении и освобождении защищенного мьютекса задействуется один бит с применением атомарной операции проверки и переустановки вместо более сложной целочисленной операции, выполняемой при работе с быстрым мьютексом.

ПРИМЕЧАНИЕ

Код быстрого мьютекса также оптимизирован с расчетом на применение почти всех этих улучшений — в нем используется такая же атомарная операция блокировки, и объект события фактически является объектом шлюза (хотя при выводе дампа типа в отладчике ядра по-прежнему будет показана структура объекта события; это просто ложь в угоду совместимости). Но быстрые мьютексы все еще повышают IRQL-уровень вместо использования защищенных областей.

Поскольку до появления Windows 2003 флаг, ответственный за отключение специальной доставки APC-вызова в режиме ядра (и за функционирование защищенной области), добавлен не был, многие драйверы не пользуются защищенными мьютексами. Такой подход способствует повышению совместимости с ранними версиями Windows, что требует от перекомпилированных драйверов использования только быстрых мьютексов. Но во внутреннем коде ядра Windows почти все использовавшиеся быстрые мьютексы были заменены защищенными мьютексами, поскольку эти два мьютекса имеют идентичную семантику и легко могут заменять друг друга.

Еще одной проблемой, связанной с защищенными мьютексами, была функция ядра KeAreApcsDisabled. До выхода Windows Server 2003 эта функция показы-

вала, отключены ли обычные APC-вызовы, проверяя, запущен ли код внутри критического раздела. В Windows Server 2003 эта функция была изменена с тем, чтобы показывать, находился ли исполняемый код в критической или защищенной области, изменяя свою работу, и для того, чтобы также вернуть TRUE, если специальные APC-вызовы режима ядра также выключены.

Поскольку есть ряд операций, которые не должны выполняться драйверами при выключенных специальных APC-вызовах режима ядра, вполне разумно будет вызвать функцию KeGetCurrentIrql для проверки, соответствует ли IRQL уровню APC или нет, что является единственным способом выключения специальных APC-вызовах режима ядра. Но, поскольку диспетчер памяти пользуется защищенными мьютексами, эта проверка дает отрицательный результат, потому что защищенные мьютексы не повышают IRQL-уровень. Вместо этого драйверы должны вызывать для этой цели функцию KeAreAllApcsDisabled. Эта функция проверяет выключение специальных APC-вызовов режима ядра и (или) нахождение IRQL на APC-уровне, что является надежным способом определения наличия как защищенных, так и быстрых мьютексов.

Ресурсы исполняющей системы

Ресурсы исполняющей системы являются механизмом синхронизации, поддерживающим совместный и исключающий доступ. Подобно быстрым мьютексам, перед их получением требуется выключение доставки обычных APC-вызовов режима ядра. Они также построены на объектах диспетчера, которые используются только тогда, когда между ними возникает конкуренция. Ресурсы исполняющей системы используются по всей системе, особенно в драйверах файловой системы, поскольку такие драйверы склонны к продолжительным периодам ожидания, во время которых должны быть по-прежнему до определенной степени разрешены операции ввода-вывода (например, для чтения).

Потоки, ожидающие получения ресурса исполняющей системы для совместного доступа, ждут семафора, связанного с ресурсом, а потоки, ожидающие получения ресурса исполняющей системы для исключающего доступа, ждут события. Семафор с неограниченным счетчиком используется для потоков, ожидающих совместного доступа, поскольку все они могут быть разбужены и им может быть предоставлено право доступа к ресурсу, когда исключающий держатель ресурса освобождает этот ресурс, путем простой выдачи сигнала семафора. Когда поток ждет исключающего доступа к ресурсу, который в настоящий момент находится в чьем-нибудь владении, он находится в ожидании объекта события синхронизации, поскольку только один из ожидающих потоков будет разбужен, когда поступит сигнал о событии. В предыдущем разделе, посвященном событиям синхронизации, упоминалось, что некоторые операции, не ожидающие события, могут стать причиной повышения уровня приоритета: этот сценарий разыгрывается, когда используются ресурсы исполняющей системы, что является одной из причин, почему они также, подобно мьютексам, отслеживают принадлежность.

Благодаря той гибкости, которая предоставляется совместным и исключающим доступом, существует несколько функций для получения ресурсов: ExAcquireResourceSharedLite, ExAcquireResourceExclusiveLite, ExAcquireSharedStarveExclusive, ExAcquireShareWaitForExclusive. Эти функции документированы в WDK.

ЭКСПЕРИМЕНТ: ВЫВОД СПИСКА ПОЛУЧЕННЫХ РЕСУРСОВ ИСПОЛНЯЮЩЕЙ СИСТЕМЫ

Команда отладчика ядра !locks ведет поиск объектов ресурсов исполняющей системы в пуле страничной памяти и выводит дамп их состояния. По умолчанию команда выводит список только тех ресурсов исполняющей системы, которые на тот момент имели владельцев, но при использовании ключа –d выводится список всех ресурсов исполняющей системы. Частичный вывод этой команды имеет следующий вид:

```
lkd> !locks
**** DUMP OF ALL RESOURCE OBJECTS ****
KD: Scanning for held locks.

Resource @ 0x89929320    Exclusively owned
    Contention Count = 3911396
     Threads: 8952d030-01<*>
KD: Scanning for held locks.....................................

Resource @ 0x89da1a68    Shared 1 owning threads
     Threads: 8a4cb533-01<*> *** Actual Thread 8a4cb530
```

Обратите внимание на то, что в счетчике конкуренции (contention count), извлекаемом из структуры ресурса, записывается количество таймерных потоков, пытавшихся получить ресурс и вынужденных ждать, поскольку им в настоящее время кто-то уже владеет.

Изучить подробность конкретного ресурсного объекта, включая поток, владеющий ресурсом, и любые, ожидающие его потоки, можно, указав ключ –v и адрес ресурса:

```
lkd> !locks -v 0x89929320

Resource @ 0x89929320    Exclusively owned
    Contention Count = 3913573
     Threads: 8952d030-01<*>
     THREAD 8952d030  Cid 0acc.050c  Teb: 7ffdf000 Win32Thread: fe82c4c0 RUNNING on
processor 0
     Not impersonating
     DeviceMap                 9aa0bdb8
     Owning Process            89e1ead8       Image:         windbg.exe
     Wait Start TickCount      24620588       Ticks: 12 (0:00:00:00.187)
     Context Switch Count      772193
     UserTime                  00:00:02.293
     KernelTime                00:00:09.828
     Win32 Start Address windbg (0x006e63b8)
     Stack Init a7eba000 Current a7eb9c10 Base a7eba000 Limit a7eb7000 Call 0
     Priority 10 BasePriority 8 PriorityDecrement 0 IoPriority 2 PagePriority 5
Unable to get context for thread running on processor 1, HRESULT 0x80004001
1 total locks, 1 locks currently held
```

Пуш-блокировки

Еще одним механизмом синхронизации, построенным на шлюзовых объектах, являются пуш-блокировки. Как и защищенные мьютексы, они ждут шлюзовый объект только тогда, когда существует конкуренция блокировки. Они предлагают некоторые преимущества по сравнению с защищенным мьютексом. Преимущества заключаются в том, что пуш-блокировки могут быть получены в режиме совместного использования или в исключающем режиме. Но основным преимуществом является их размер: объект ресурса занимает 56 байт, а размер пуш-блокировки равен размеру указателя. К сожалению, пуш-блокировки не документированы в WDK, и поэтому они зарезервированы для использования операционной системой (хотя API-функции экспортируются, и внешние драйверы их используют).

Есть два типа пуш-блокировок: обычные и осведомленные о кэш-памяти. Обычные пуш-блокировки требуют в хранилище места не больше, чем указатель (4 байта на 32-разрядных системах и 8 байт на 64-разрядных системах). Когда поток получает обычную пуш-блокировку, код пуш-блокировки помечает ее как принадлежащую этому потоку, если только она в данный момент не находится в чьем-нибудь владении. Если кто-нибудь владеет пуш-блокировкой в исключающем режиме или если поток желает получить ее в исключительное владение, а пуш-блокировка находится в совместном использовании, поток размещает блок ожидания в стеке потока, инициализирует шлюзовый объект в блоке ожидания и добавляет блок ожидания к списку ожидания, который связан с пуш-блокировкой. Когда поток освобождает пуш-блокировку, он будит ожидающий поток, если таковой имеется, путем ввода объекта события, имеющегося в блоке ожидания потока, в сигнальное состояние.

Поскольку пуш-блокировка имеет размер указателя, она фактически содержит различные биты, описывающие ее состояние. Значения этих битов изменяются в зависимости от перехода пуш-блокировки из конкурирующего в неконкурирующее состояние. В своем исходном состоянии пуш-блокировка содержит следующую структуру:

- один бит блокировки, установленный в 1, если блокировка получена;
- один бит ожидания, установленный в 1, если в отношении блокировки есть конкуренция и кто-то ее ожидает;
- один бит пробуждения, установленный в 1, если блокировка предоставлена потоку и нужна оптимизация списка ожидающих потоков;
- один бит совместного использования, установленный в 1, если пуш-блокировка используется совместно и в данный момент получена более чем одним потоком;
- 28 (на 32-разрядных Windows) или 60 (на 64-разрядных Windows) битов счетчика совестного использования, в котором содержится количество потоков, получивших пуш-блокировку.

Как уже рассматривалось, когда поток получает пуш-блокировку в исключающем режиме в тот момент, когда она уже получена либо несколькими читающими, либо одним записывающим потоком, ядро размещает блок ожидания пуш-блокировки. Структура значения самой пуш-блокировки изменяется. Теперь биты

счетчика совместного использования становятся указателем на блок ожидания. Поскольку этот блок ожидания размещен в стеке, а файлы заголовков содержат специальную директиву выравнивания, заставляя его выровняться по 16-битному формату, последние 4 бита структуры любого блока ожидания пуш-блокировки будут всегда содержать нули. Поэтому данные биты игнорируются с целью разыменования указателя; взамен показанные ранее 4 бита объединяются со значением указателя. Поскольку в результате этого выравнивания удаляются биты счетчика совместного использования, теперь этот счетчик хранится в блоке ожидания.

Пуш-блокировка, осведомленная о кэш-памяти, добавляет к обычной (основной) пуш-блокировке уровни путем размещения пуш-блокировки для каждого процессора в системе и связывания ее с пуш-блокировкой, осведомленной о кэш-памяти. Когда потоку нужно получить кэш-блокироку, осведомленную о кэш-памяти, для совместного доступа, он просто получает пуш-блокировку, размещенную для его текущего процессора в режиме совместного использования; для получения исключающей кэш-блокироки, осведомленной о кэш-памяти, поток получает пуш-блокировку для каждого процессора в исключающем режиме.

Кроме существенно меньших следов, оставляемых в памяти, одним из наибольших преимуществ таких пуш-блокировок над ресурсами исполняющей системы является то, что в отсутствие конкуренции они не требуют для своего получения или освобождения длинных учетных и целочисленных операций. Поскольку по размеру они равны указателю, ядро для выполнения этих задач может использовать атомарные инструкции центрального процессора (инструкция lock cmpxchg атомарно проводит сравнение и замену старой блокировки новой). Если операция атомарного сравнения и замены не удается, блокировка содержит значение, неожиданное для вызывающего модуля (вызывающие модули обычно ожидают, что блокировка не используется или получена в режиме совместного использования), и затем делается вызов более сложной версии, используемой при возникновении конкуренции. Для дальнейшего повышения производительности ядро выставляет функциональные возможности пуш-блокировки в виде встроенных функций, что означает отсутствие каких-либо вызовов функций в процессе бесконкурентного получения — в каждую функцию непосредственно вставляется ассемблерный код. Это немного увеличивает размер кода, но позволяет избежать замедлений, связанных с вызовом функции. И наконец, в пуш-блокировках используются обычно несколько алгоритмических приемов с целью избежать сопровождений блокировки (когда несколько потоков с одним и тем же приоритетом ожидают блокировку и выполняется совсем небольшой объем фактической работы), и они также самооптимизируются: список потоков, ожидающих пуш-блокировку, будет периодически перестроен для более справедливого поведения при освобождении пуш-блокировки.

ОПРЕДЕЛЕНИЕ ВЗАИМНЫХ БЛОКИРОВОК С ПОМОЩЬЮ ВЕРИФИКАТОРА ДРАЙВЕРОВ

Взаимная блокировка является проблемой синхронизации, возникающая, когда два потока или процессора удерживают ресурсы, необходимые друг другу, и никогда не уступают то, чем владеют. Эта ситуация может вылиться в зависание системы или процесса. Верификатор драйверов Driver Verifier имеет настройку для проверки взаимных блокировок при использовании спин-блокировок, а также быстрых и простых мьютексов.

Области, в которых используются пуш-блокировки, включают диспетчер объектов, где они защищают глобальные структуры данных диспетчера объектов и дескрипторы безопасности объекта, и диспетчер памяти, где их аналоги, осведомленные о кэш-памяти, используются для защиты структуры данных Address Windowing Extension (AWE).

Критические разделы

Критические разделы являются одним из основных примитивов синхронизации, который Windows предоставляет приложениям пользовательского режима помимо примитивов синхронизации, основанными на работе в режиме ядра. Критические разделы и другие примитивы пользовательского режима, как позже станет понятно, имеют одно главное преимущество над своими аналогами режима ядра, которые постоянно переходят в режим ядра и обратно в тех случаях, когда блокировка осуществляется в отсутствие конкуренции (что обычно занимает 99 % времени, если не больше). Но в случае конкуренции по-прежнему требуется вызов ядра, потому что это единственная часть системы, которая способна выполнять сложное пробуждение и применять логику диспетчеризации, которые необходимы, чтобы объекты работали.

Критические разделы могут оставаться в пользовательском режиме путем использования локального бита для обеспечения основной логики исключающей блокировки, которая во многом похожа на логику спин-блокировки. Если бит содержит 0, критический раздел может быть получен, и владелец установит этот бит в 1. Эта операция не требует вызова ядра, но использует рассмотренные ранее операции взаимоблокировки центрального процессора. Освобождение критического раздела осуществляется аналогичным образом, когда с помощью операции взаимоблокировки бит состояния изменяется с 1 на 0. С другой стороны, как вы уже могли догадаться, когда бит уже имеет значение 1 и другой вызывающий код пытается получить критический раздел, должно быть вызвано ядро, чтобы поместить поток в состояние ожидания. И наконец, поскольку критические разделы не являются объектами ядра, у них имеется ряд ограничений. Одно из основных ограничений заключается в том, что вы не можете получить дескриптор ядра, описывающий критический раздел, в силу этого к критическому разделу не может быть применено обеспечение безопасности, присваивание имени и другие функциональные возможности диспетчера объектов. Два процесса не могут использовать один и тот же критический раздел для координации своих действий, а также не могут использовать дублирование и наследование.

Ресурсы пользовательского режима

Ресурсы пользовательского режима также предоставляют более тонкие механизмы блокировки, чем примитивы ядра. Ресурс могут быть получен для режима совместного использования или для режима исключающего использования, позволяя ему работать в режиме блокировки для нескольких читающих потоков (при совместном использовании), в режиме блокировки для единственного записывающего потока (при исключающем использовании) для таких структур данных, как базы данных. Когда происходит получение ресурса в режиме совместного использования и другие потоки пытаются получить то же самый ресурс, не требуется никаких

переходов в режим ядра, поскольку ни один из потоков не будет ожидающим. Это потребуется только когда поток попытается получить ресурс для исключающего доступа или когда ресурс уже заблокирован исключающим владельцем.

Для использования тех же механизмов диспетчеризации и синхронизации, которые вы видели в ядре, ресурсы фактически используют существующие примитивы ядра. Структура данных ресурса (RTL_RESOURCE) содержит дескрипторы объекта мьютекса ядра, а также объекта семафора ядра. При получении ресурса в исключающее использование более чем одним потоком ресурс использует мьютекс, поскольку он разрешает наличие только одного владельца. При получении ресурса в совместное использование более чем одним потоком ресурс использует семафор, поскольку он разрешает наличие нескольких пользователей. Этот уровень детализации обычно скрыт от программиста, и эти внутренние объекты никогда не будут использоваться напрямую.

Изначально ресурсы были реализованы для поддержки администратора учетных данных в системе защиты — Security Account Manager (SAM) — и для стандартных приложений в Windows API не выставляются. Для выставления подобных примитивов блокировки через документированные API-функции в Windows Vista были реализованы гибкие блокировки чтения-записи — Slim Reader-Writer Locks (SRW Locks), описываемые далее, хотя некоторые системные компоненты по-прежнему используют механизм ресурсов.

Условные переменные

Условные переменные обеспечивают присущую Windows реализацию синхронизации набора потоков, ожидающих конкретного результата проверки условия. Хотя эта операция была возможна и с применением других методов синхронизации в пользовательском режиме, *атомарного* механизма для проверки результата проверки условия и для начала ожидания изменения этого результата не существовало. Это потребовало использования в отношении таких фрагментов кода этой дополнительной синхронизации.

Поток пользовательского режима инициализирует условную переменную путем вызова функции InitializeConditionVariable для установки ее исходного состояния. Когда ему нужно инициировать ожидание, связанное с этой переменной, он вызывает функцию SleepConditionVariableCS, которая использует критический раздел (который поток должен инициализировать) для ожидания изменений переменной. После изменения переменной устанавливающий поток должен использовать функцию WakeConditionVariable (или функцию WakeAllConditionVariable)[1]. В результате этого вызова освобождается критический раздел либо одного, либо всех ожидающих потоков, в зависимости от того, которая из этих функция была использована.

До условных переменных для отправки сигнала об изменении переменной, например состояния рабочей очереди, часто использовалось либо *уведомительное событие*, либо *синхронизирующее событие* (в Windows API они называются *автоматическим перезапуском* — auto-reset, или *ручным перезапуском* — manual-reset). Ожидание изменения требует получения, а затем освобождения критического раздела, сопровождаемого ожиданием события. После ожидания критический

[1] Автоматического механизма определения изменения не существует.

раздел должен быть получен повторно. Во время этих серий получений и освобождений у потока может быть переключен контекст, вызывая проблемы, если один из потоков называется *PulseEvent*[1]. При использовании условных переменных получение критического раздела может быть поддержано приложением во время вызова функции leepConditionVariableCS, и он может быть освобожден только после выполнения работы. Это делает код записи рабочей очереди (и подобных ей реализаций) более простым и предсказуемым.

Внутри системы условные переменные могут рассматриваться как порт существующих алгоритмов пуш-блокировок, имеющихся в режиме ядра, с дополнительным усложнением в виде получения и освобождения критических разделов в API-функции SleepConditionVariableCS. Условные переменные по размеру равны указателям (точно так же, как и пуш-блокировки), избегают использования диспетчера, автоматически оптимизируют во время операций ожидания список ожиданий и защищают от сопровождений блокировки. Кроме того, условные переменные полностью используют события с ключом, а не обычный объект события, который бы использовался разработчиками по своему усмотрению, что еще больше оптимизирует код даже в случаях возникновения конкуренции.

Гибкие блокировки чтения-записи (Slim Reader-Writer Locks)

Хотя условные переменные являются механизмом синхронизации, они не являются всецело примитивными объектами блокировки. Как вы уже поняли, они все же зависят от блокировки критического раздела, при получении и освобождении которого используется стандартный диспетчер объектов событий. Поэтому путешествие через режим ядра может все же произойти, и вызывающим модулям все же требуется инициализация большого объекта критического раздела. Если условные переменные имеют множество общих черт с пуш-блокировками, то и гибкие блокировки чтения-записи (Slim Reader-Writer Locks, SRW-блокировки) устроены сходным образом. Они также имеют размер указателей, используют атомарные операции для получения и освобождения, перестраивают свои списки ожиданий, защищают от сопровождения блокировок и могут быть получены как в режиме совместного использования, так и в исключающем режиме. Но некоторые отличия от пуш-блокировок все же есть: в частности, SRW-блокировки не могут быть «модернизированы» или конвертированы из режима совместного использования в режим исключающего использования или наоборот. Кроме того, они не могут быть получены рекурсивно. И наконец, SRW-блокировки являются исключающими по отношению к коду пользовательского режима, в то время как пуш-блокировки являются исключающими по отношению к коду в режиме ядра. Эти два вида блокировок не могут быть совместно использованы или видны из одного уровня в другом.

SRW-блокировки не только могут полностью заменить критические разделы в коде приложения, но также еще и предлагают функциональные возможности, допускающие наличие нескольких читающих и одного записывающего модуля. SRW-блокировки должны сначала быть инициализированы с помощью функ-

[1] Аналогичную проблему для событий с ключом решили за счет принуждения к ожиданию сигнализирующего потока при отсутствии ожидающих потоков.

ции InitializeSRWLock, после чего они могут быть получены или освобождены либо в режиме исключающего использованиям, либо в режиме общего использования с помощью соответствующих API-функций: AcquireSRWLockExclusive, ReleaseSRWLockExclusive, AcquireSRWLockShared и ReleaseSRWLockShared.

ПРИМЕЧАНИЕ

В отличие от большинства API-функций Windows APIs, функции SRW-блокировки не возвращают управление с каким-нибудь значением. Вместо этого, если блокировка не может быть получена, они генерируют исключения. При этом становится совершенно очевидным тот факт, что получить блокировку не удалось, и поэтому выполнение кода, который пытается это сделать, будет завершено, вместо того чтобы дать ему потенциальную возможность разрушить пользовательские данные.

Имеющиеся в Windows SRW-блокировки не дают предпочтений ни коду, ведущему чтение, ни коду, ведущему запись, стало быть, их действия в любом случае должны быть одинаковыми. Это делает их хорошей заменой критических разделов, являющихся механизмами синхронизации только для кода, ведущего чтение, или для исключительного доступа к данным, кроме того, они предоставляют оптимизированную альтернативу ресурсам. Если бы SRW-блокировки были оптимизированы для кода, ведущего чтение, то они были бы самыми примитивными блокировками, применяемыми только для исключающего доступа, но это не так. В результате конструкция представленного ранее механизма условной переменной также позволяет использовать SRW-блокировки вместо критических разделов, для чего используется API-функция SleepConditionVariableSRW. И наконец, SRW-блокировки используют вместо стандартных объектов событий события с ключом. Это позволяет сочетаниям условных переменных и SRW-блокировок стать масштабируемыми механизмами синхронизации, имеющими размеры указателей с весьма скромным количеством переходов в режим ядра исключительно в условиях конкуренции, оптимизированными на меньшие затраты времени и памяти для пробуждения и установку благодаря использованию событий с ключом.

Единовременная инициализация

Возможность гарантировать атомарное выполнение кодового фрагмента, отвечающего за выполнение определенной задачи инициализации (например, выделения памяти, инициализации определенных переменных или даже создания требуемых объектов), является типичной задачей многопотокового программирования. Во фрагменте кода, который может быть вызван одновременно несколькими потоками[1], есть несколько способов попытки обеспечения корректного, атомарного и однозначного выполнения задач инициализации.

По данному сценарию Windows реализует то, что называется *init once*, или *one-time initialization* (что также во внутренних документах называется *run once initialization*), то есть проводит *единовременную инициализацию*. Этот механизм допускает как синхронное выполнение определенного фрагмента кода (ожидание со стороны других потоков момента завершения инициализации), так и его асинхронное выполнение (Предоставление возможности другим потокам попытки

[1] Хорошим примером может послужить процедура DllMain, инициализирующая DLL-библиотеку.

проведения своей собственной инициализации и вступление в соревнование). Логика асинхронного выполнения будет объяснена после рассмотрения механизма синхронного выполнения.

При синхронном выполнении разработчик создает фрагмент кода, который обычно будет выполняться после двойной проверки глобальной переменной в специальной функции. Любая информация, которая нужна этой процедуре, может быть передана через переменную параметра parameter, которую принимает процедура единовременной инициализации. Любая выходная информация возвращается через переменную контекста context. Состояние самой инициализации возвращается как булево значение — Boolean. Все, что должен сделать разработчик для обеспечения надлежащего выполнения — это вызвать функцию InitOnceExecuteOnce с parameter, context и указатель на функцию единовременного запуска после инициализации объекта INIT_ONCE с помощью API-функции InitOnceInitialize. Обо всем остальном позаботится система.

Для приложений, которым вместо этого нужно использовать асинхронную модель, потоки вызывают функцию InitOnceBeginInitialize и получают BOOLEAN-значение отложенного состояния — pending status — и описанное ранее значение context. Если значение pending status равно FALSE, значит, инициализация уже состоялась, и поток использует в качестве результата значение context. Вернуть FALSE может и сама функция, что будет означать неудачный исход инициализации. Но если в pending status возвращается значение TRUE, поток должен вступить в соревнование, чтобы первым создать объект. Последующий код выполняет все, что требуется от задачи инициализации, например занимается созданием объектов или выделением памяти. Когда эта работа будет сделана, поток вызывает функцию InitOnceComplete с результатом работы в виде значения context, и получает BOOLEAN-значение status. Если значение status равно TRUE, значит, поток выиграл соревнование, и объект, который создан или распределен, станет глобальным объектом. Теперь поток может сохранить этот объект или вернуть его вызывавшему модулю, в зависимости от того, что применимо.

При более сложном сценарии, если status имеет значение FALSE, это означает, что поток проиграл соревнование. Поток должен произвести откат всей проделанной работы, например отменить удаление объектов или освобождение памяти, а затем снова вызвать функцию InitOnceBeginInitialize. Но вместо запроса на старт соревнований, как он это первоначально сделал, он использует флаг проведения всего лишь проверки возможности единовременной инициализации INIT_ONCE_CHECK_ONLY, зная о своих потерях, и запрашивает вместо прежних данных значение context победителя (например, тех созданных победителем объектов или той выделенной им памяти). Здесь уже возвращается другое значение status, которое может быть TRUE и означать, что context имеет действующее значение, которое может быть использовано или возвращено вызывавшему модулю, или может быть FALSE, и означать, что инициализация прошла неудачно и никто в данный момент не сможет выполнить работу в условиях дефицита памяти.

В обоих случаях механизм для единовременной инициализации подобен механизму, используемому для условных переменных и SRW-блокировок. Структура единовременной инициализации имеет размер указателя, и в случае отсутствия конкуренции используются встроенные ассемблерные версии кода получения-освобождения SRW-блокировки. В случае возникновения конкуренции используются события с ключом (когда механизм используется в синхронном режиме),

а остальные потоки должны ждать инициализации. В случае использования асинхронного режима блокировки используются совместно, поэтому инициализация может одновременно осуществляться сразу несколькими потоками.

Системные рабочие потоки

При инициализации системы Windows создает в процессе System несколько потоков, которые называются *системными рабочими потоками* и существуют исключительно для выполнения работы по поручению других потоков. Во многих случаях потокам, выполняющимся на уровне DPC/dispatch, требуется выполнять функции, которые могут работать только на более низком уровне IRQL. Например, DPC-процедуре, выполняемой в произвольном контексте потока (для выполнения DPC может быть захвачен любой поток, имеющийся в системе) на IRQ-уровне DPC/dispatch, может понадобиться доступ к выгружаемому пулу памяти или ожидание объекта диспетчера, используемого для синхронизации выполнения с потоком приложения. Поскольку DPC-процедура не может понизить IRQL, она должна передать обработку потоку, в котором выполнение происходит на IRQL-уровне, находящемся ниже уровня DPC/dispatch.

Некоторые драйверы устройств и компоненты исполняющей системы создают свои собственные потоки, предназначенные для вычислительной работы на пассивном (passive) уровне. Но большинство все же использует вместо них системные рабочие потоки, что исключает излишнюю диспетчеризацию и неоправданный расход памяти, связанные с наличием в системе дополнительных потоков. Компонент исполняющей системы запрашивает службы системных рабочих потоков, вызывая функции исполняющей системы ExQueueWorkItem или IoQueueWorkItem. Драйверы устройств должны использовать только последнюю функцию (это связывает рабочий элемент с объектом Device, предусматривая более высокий уровень возможности учета ситуации и управления сценариями, в которых драйвер выгружается, в то время как рабочий элемент является активным). Эти функции помещают *рабочий элемент* (work item) в объект очереди диспетчера, где потоки ищут работу.

API-функции IoQueueWorkItemEx, IoSizeofWorkItem, IoInitializeWorkItem и IoUninitializeWorkItem работают схожим образом, но они создают связь с имеющимся в драйвере объектом Driver или с одним из имеющимся в нем объектом Device.

Рабочие элементы включают в себя указатель на процедуру и параметр, который поток передает процедуре при обработке рабочего элемента. Процедура реализуется в драйвере устройства или в компонент исполняющей системы, требующем выполнения на уровне passive. Например, DPC-процедура, которая должна ожидать объект диспетчера, может инициализировать рабочий элемент, указывающий на процедуру в драйвере, которая ожидает объект диспетчера и, возможно, указывает на объект. На каком-то этапе системный рабочий поток удалит рабочий элемент из своей очереди и выполнит процедуру драйвера. При завершении процедуры драйвера системный рабочий поток проводит проверку на наличие каких-либо необработанных рабочих элементов. Если таковых больше нет, системный рабочий поток блокируется до тех пор, пока в очередь не будет помещен рабочий элемент. Когда системный рабочий поток обрабатывает свой рабочий элемент, DPC-процедура может уже завершить, а может еще и не завершить свое выполнение.

Существует три типа системных рабочих потоков:

- *Отложенные рабочие потоки* (delayed worker threads) выполняются с приоритетом 12, обрабатывают рабочие элементы, не считающиеся критичными по времени, и в режиме ожидания рабочих элементов допускают выгрузку своих стеков в файл подкачки. Диспетчер объектов использует отложенные рабочие элементы для осуществления отложенного удаления объекта, которое удаляет объекты ядра после того, как они были спланированы на освобождение.
- *Критические рабочие потоки* (critical worker threads) выполняются с приоритетом 13, обрабатывают критичные по времени рабочие элементы и на системах Windows Server постоянно содержат свои стеки в физической памяти.
- Единственный *сверхкритический рабочий поток* (hypercritical worker thread) выполняется с приоритетом 15 и также содержит свой стек в памяти. Диспетчер процессов использует сверхкритический рабочий элемент для выполнения «подчищающей» функции потока, которая освобождает завершенные потоки.

Количество отложенных и критических рабочих потоков, созданных функцией исполняющей системы ExpWorkerInitialization, которая вызывается на ранней стадии процесса загрузки, зависит от количества памяти, имеющейся в системе, и от того, является ли система сервером. В табл. 3.22 показано начальное количество потоков, созданных при настройках по умолчанию. Функции ExpInitializeWorker с помощью параметров AdditionalDelayedWorkerThreads и AdditionalCriticalWorkerThreads в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Executive можно указать о необходимости создания до 16 дополнительных отложенных и до 16 дополнительных критических рабочих потоков.

Таблица 3.22. Исходное количество системных рабочих потоков

Тип рабочей очереди	Количество потоков по умолчанию
Отложенные	7
Критические	5
Сверхкритические	1

Исполняющая система старается привести в соответствие количество критических рабочих потоков изменяющейся во время работы системы рабочей нагрузке. Ежесекундно функция исполняющей системы ExpWorkerThreadBalanceManager определяет, должна ли она создать новый критический рабочий поток. Критические рабочие потоки, созданные функцией ExpWorkerThreadBalanceManager, называются динамическими рабочими потоками, и перед созданием такого потока должны быть соблюдены следующие условия:

- В критической рабочей очереди есть рабочие элементы.
- Количество пассивных критических рабочих потоков (либо заблокированных в ожидании рабочих элементов, либо заблокированных на объектах диспетчера на время выполнения рабочей процедуры) должно быть меньше количества процессоров в системе.
- Количество динамических рабочих потоков не превышает шестнадцати.

Выход из динамических рабочих потоков осуществляется после 10 минут бездействия. Таким образом, если это продиктовано рабочей нагрузкой, исполняющая система может создать до 16 динамических рабочих потоков.

ЭКСПЕРИМЕНТ: ВЫВОД СПИСКА СИСТЕМНЫХ РАБОЧИХ ПОТОКОВ

Для просмотра списка системных рабочих потоков, классифицированных по типу, можно воспользоваться командой отладчика ядра !exqueue:

```
lkd> !exqueue
Dumping ExWorkerQueue: 820FDE40
**** Critical WorkQueue( current = 0 maximum = 2 )
THREAD 861160b8  Cid 0004.001c  Teb: 00000000 Win32Thread: 00000000 WAIT
THREAD 8613b020  Cid 0004.0020  Teb: 00000000 Win32Thread: 00000000 WAIT
THREAD 8613bd78  Cid 0004.0024  Teb: 00000000 Win32Thread: 00000000 WAIT
THREAD 8613bad0  Cid 0004.0028  Teb: 00000000 Win32Thread: 00000000 WAIT
THREAD 8613b828  Cid 0004.002c  Teb: 00000000 Win32Thread: 00000000 WAIT
**** Delayed WorkQueue( current = 0 maximum = 2 )
THREAD 8613b580  Cid 0004.0030  Teb: 00000000 Win32Thread: 00000000 WAIT
THREAD 8613b2d8  Cid 0004.0034  Teb: 00000000 Win32Thread: 00000000 WAIT
THREAD 8613c020  Cid 0004.0038  Teb: 00000000 Win32Thread: 00000000 WAIT
THREAD 8613cd78  Cid 0004.003c  Teb: 00000000 Win32Thread: 00000000 WAIT
THREAD 8613cad0  Cid 0004.0040  Teb: 00000000 Win32Thread: 00000000 WAIT
THREAD 8613c828  Cid 0004.0044  Teb: 00000000 Win32Thread: 00000000 WAIT
THREAD 8613c580  Cid 0004.0048  Teb: 00000000 Win32Thread: 00000000 WAIT
**** HyperCritical WorkQueue( current = 0 maximum = 2 )
THREAD 8613c2d8  Cid 0004.004c  Teb: 00000000 Win32Thread: 00000000 WAIT
```

Глобальные флаги Windows

В Windows есть набор флагов, сохраненных в общесистемной глобальной переменной NtGlobalFlag, с помощью которой в операционной системе включается поддержка различной внутренней отладки, трассировки и проверки. Во время загрузки системы системная переменная NtGlobalFlag инициализируется из параметра GlobalFlag раздела реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager. По умолчанию значение этого параметра реестра равно нулю, так что, скорее всего, на вашей системе глобальные флаги не используются. Кроме того, у каждого образа есть набор глобальных флагов, с помощью которых также включается код внутренней трассировки и проверки (хотя поразрядная структура этих флагов полностью отличается от структуры общесистемных глобальных флагов).

К счастью, инструментарий отладки содержит утилиту под названием Gflags.exe, которую можно использовать для просмотра и изменения системных глобальных флагов (либо в реестре, либо в работающей системе), а также глобальных флагов образов. У утилиты Gflags есть как интерфейс командной строки, так и графический пользовательский интерфейс. Для просмотра флагов командной строки нужно набрать команду gflags /?. Если утилита запускается без ключей, выводится диалоговое окно, показанное на рис. 3.28.

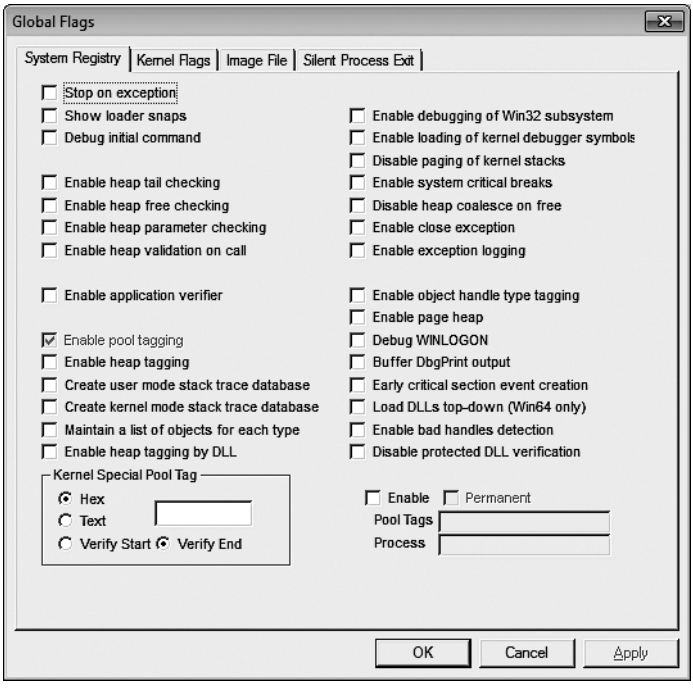


Рис. 3.28. Установка параметров отладки системы с помощью утилиты Gflags

На странице System Registry (Реестр системы) можно настроить переменную реестра, а на странице Kernel Flags (Флаги ядра) можно настроить текущее значение переменной в системной памяти.

На странице Image File (Файл образа) требуется набрать в поле имя файла исполняемого образа. Этот вариант используется для изменения набора глобальных флагов, применяемых к отдельному образу (а не ко всей системе). Обратите внимание, что флаги на рис. 3.29 отличаются от флагов операционной системы, показанных на рис. 3.28.

ЭКСПЕРИМЕНТ: ПРОСМОТР И УСТАНОВКА NTGLOBALFLAG

Для просмотра состояния переменной ядра NtGlobalFlag и установки ее значения можно воспользоваться командой отладчика ядра !gflag. Эта команда выводит список всех установленных флагов. Для вывода полного списка поддерживаемых глобальных флагов можно воспользоваться командой !gflag -?. ■

Усовершенствованный вызов локальных процедур

Всем современным операционным системам нужен механизм для безопасного переноса данных между несколькими процессами в пользовательском режиме, а также между службой в режиме ядра и клиентами в пользовательском режиме.

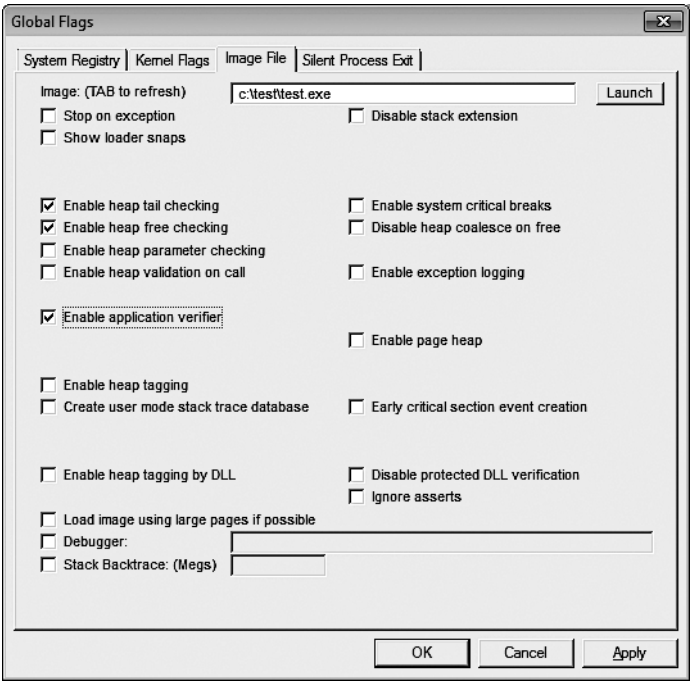


Рис. 3.29. Установка глобальных флагов образа с помощью утилиты Gflags

Обычно для переносимости используются такие UNIX-механизмы, как почтовые слоты, файлы, именованные каналы и сокеты, но есть и такие разработчики, которые для графических приложений используют оконные сообщения. В Windows реализуется внутренний IPC-механизм — усовершенствованный вызов локальных процедур (Advanced Local Procedure Call, ALPC), обладающий высокой скоростью работы, масштабируемостью и средством обеспечения передачи сообщений произвольного размера. Хотя этот механизм является внутренним и поэтому недоступен сторонним разработчикам, ALPC широко используется в различных частях Windows:

- Windows-приложения, использующие вызов удаленных процедур (remote procedure call, RPC), относящийся к документированному API, опосредованно задействуют ALPC, когда определяют локальный RPC-вызов через ncalrpc-транспортировку, форму RPC, используемую для связи между процессами на одной и той же системе. ALPC также используется RPC-вызовами режима ядра, которые задействуются сетевым стеком.
- ALPC используется при каждом запуске процесса и (или) потока Windows, а также во время любой операции подсистемы Windows (например, при всех консольных вводах выводах) для связи с процессом подсистемы (CSRSS). Через ALPC все подсистемы связываются с диспетчером сеанса — session manager (SMSS).
- Winlogon использует ALPC для связи с процессом проверки подлинности локальной системы безопасности, LSASS.

- Монитор безопасности ссылок (компонент исполняющей системы, рассматриваемый в главе 6) использует ALPC для связи с LSASS-процессом.
- Через ALPC диспетчер электропитания и монитор электропитания пользовательского режима связываются с диспетчером электропитания режима ядра, например, при изменении яркости жидкокристаллического дисплея.
- ALPC используется системой отчета об ошибках Windows Error Reporting для получения контекстной информации от аварийных процессов.
- Среда драйверов пользовательского режима — User-Mode Driver Framework (UMDF) позволяет драйверам пользовательского режима обмениваться данными, используя ALPC.

ПРИМЕЧАНИЕ

ALPC является заменой устаревшего IPC-механизма, первоначально поставлявшегося с самыми первыми конструкциями ядра Windows NT, под названием LPC, в силу чего определенные переменные, поля и функции могут по-прежнему и сегодня ссылаться на «LPC». Следует иметь в виду, что механизм LPC теперь эмулируется на верхнем уровне ALPC для совместимости и удален из ядра, но существуют еще устаревшие системные вызовы, которые заключаются в оболочку ALPC-вызовов.

Модель подключения

Обычно ALPC-вызовы используются между серверным процессом и одним или несколькими клиентскими процессами этого сервера. ALPC-подключение может быть установлено между двумя или несколькими процессами пользовательского режима или между компонентом режима ядра и одним или несколькими процессами пользовательского режима. Для поддержания необходимого для связи состояния ALPC экспортирует один исполняемый объект, называемый объектом порта. Хотя это всего лишь один объект, на самом деле существует несколько разновидностей ALPC-портов, которые он может представлять:

- **Порт подключения к серверу** (server connection port). Именованный порт, указываемый в запросе на подключение к серверу. Клиенты могут подключаться к серверу, подключаясь к этому порту.
- **Серверный порт связи** (server communication port). Безымянный порт, используемый сервером для связи с конкретным клиентом. У сервера есть один такой порт для каждого активного клиента.
- **Клиентский порт связи** (client communication port). Безымянный порт, используемый конкретным клиентским потоком для связи с конкретным сервером.
- **Неподключенный порт связи** (unconnected communication port). Безымянный порт, который клиент может использовать для локальной связи с самим собой.

ALPC следует модели подключения и связи, которая чем-то напоминает BSD-программирование сокетов. Сначала сервер создает порт подключения к серверу (NtAlpcCreatePort), в то время как клиент предпринимает попытку подключиться к этому порту (NtAlpcConnectPort). Если сервер находится в состоянии прослушивания, он получает сообщение о запросе подключения и может принять это сообщение (NtAlpcAcceptPort). При этом создаются как клиентский, так и серверный порты связи, и каждый соответствующий конечный процесс получает дескриптор

своего порта связи. Затем сообщения пересылаются через этот дескриптор (NtAlpcSendWaitReceiveMessage), обычно в выделенном потоке, чтобы сервер мог продолжать прослушивание запросов на подключение в исходном порту подключения (если только этот сервер не рассчитан только на одного клиента).

Сервер также может отклонить подключение либо по соображениям безопасности, либо просто из-за проблем с протоколом или версиями. Поскольку клиенты могут отправить с запросом на подключение какую-нибудь свою полезную нагрузку, это обычно используется различными службами, чтобы убедиться, что с сервером связывается надлежащий или только единственный клиент. Если будут обнаружены какие-нибудь аномалии, сервер может отклонить подключение и дополнительно, вернуть нагрузку, содержащую информацию о том, почему клиент был отклонен. Эта информация позволяет клиенту предпринять корректирующее действие или воспользоваться этой информацией в целях отладки программы.

После подключения информационная структура этого подключения (обычно в виде блоба, который вскоре будет рассмотрен) сохраняет связь между различными портами, как показано на рис. 3.30.

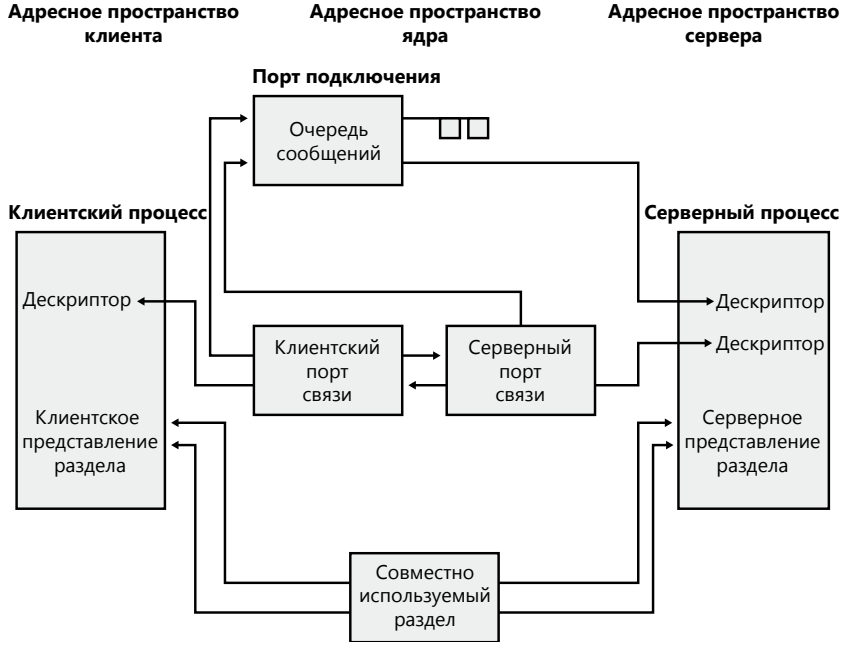


Рис. 3.30. Использование ALPC-портов

Модель сообщений

При использовании ALPC клиент и поток используют блокирующие сообщения, каждый по очереди выполняя цикл вокруг системного вызова NtAlpcSendWaitReplyPort, в котором одна из сторон отправляет запрос и ждет ответ, в то время как другая сторона делает противоположное. Но поскольку ALPC поддерживает асинхронные сообщения, у любой из сторон есть возможность не заниматься блокировкой, а выполнить вместо этого какую-нибудь другую задачу времени выпол-

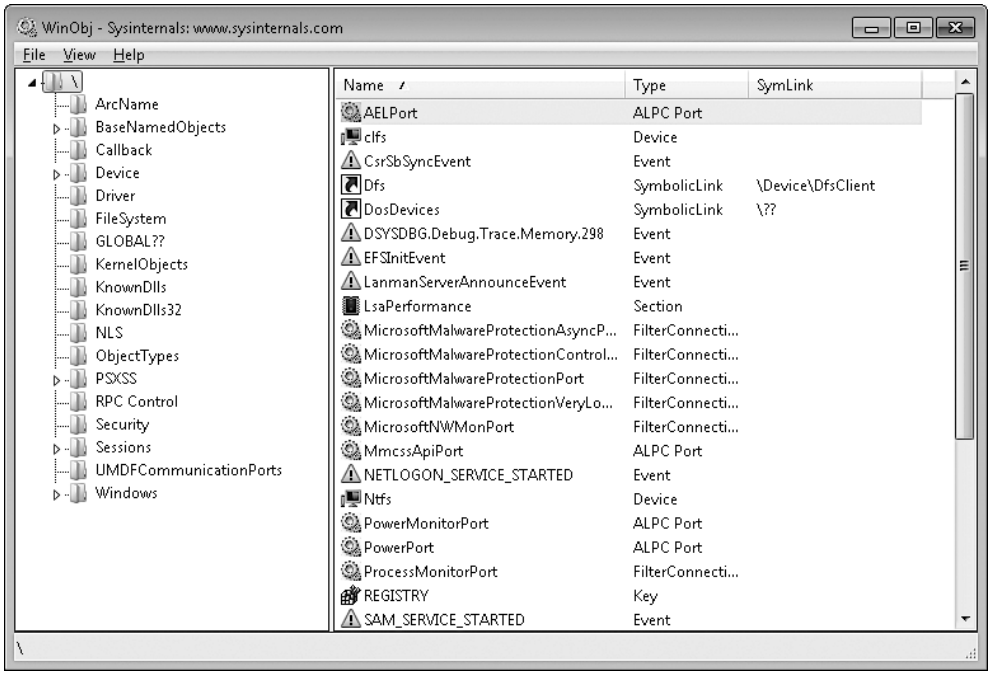
нения и проверить наличие сообщения чуть позже. ALPC поддерживает следующие три метода обмена полезной нагрузкой, отправляемой вместе с сообщением:

- Сообщение может быть отправлено другому процессу через стандартный механизм двойного буферирования, в котором ядро сохраняет копию сообщения (копируя его из процесса-источника), переключается на целевой процесс и копирует данные из буфера ядра. С целью поддержки совместимости, если используется устаревший LPC, этим способом могут быть отправлены только сообщения длиной до 256 байт, хотя у ALPC есть возможность выделить расширенный буфер для сообщений объемом примерно 64 Кбайт.
- Сообщение может быть сохранено в объекте ALPC-раздела, откуда клиент и сервер выстраивают свои видения.
- Сообщение может быть сохранено в зоне сообщений, являющейся таблицей описания памяти — Memory Descriptor List (MDL), которая поддерживает физические страницы, содержащие данные, и которая отображается в адресном пространстве ядра.

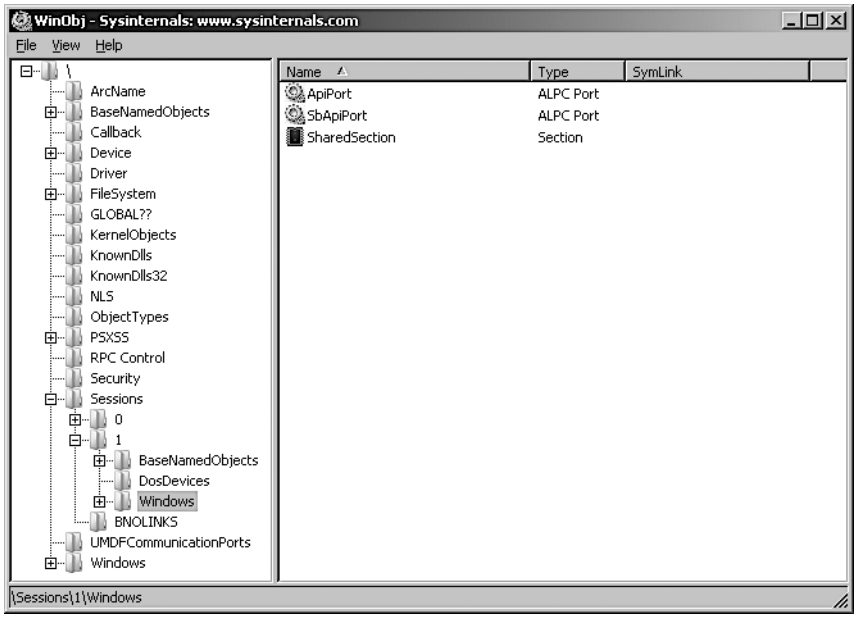
Важным побочным эффектом возможности отправки асинхронных сообщений является то, что сообщение может быть аннулировано, например, когда запрос занимает слишком много времени или пользователь указал, что он хочет отменить реализуемую им операцию. ALPC поддерживает эту возможность с помощью системного вызова NtAlpcCancelMessage.

ЭКСПЕРИМЕНТ: ПРОСМОТР ОБЪЕКТОВ ALPC-ПОРТА ПОДСИСТМЫ

Просмотреть именованные объекты ALPC-порта можно с помощью средства WinObj от Sysinternals. Запустите Winobj.exe и выберите корневой каталог. Объекты порта будут показаны с помощью значка с шестеренками.



Вы должны увидеть ALPC-порты, используемые диспетчером электропитания, диспетчером безопасности и другими внутренними службами Windows. Если вам нужно просмотреть объекты ALPC-порта, используемого RPC, вы можете выбрать каталог \RPC Control. Кроме Local RPC, одним из основных пользователей ALPC является подсистема Windows, которая использует ALPC для связи со своими DLL-библиотеками, присутствующими во всех Windows-процессах. (Подобный механизм используется также в подсистеме для UNIX-приложений.) Поскольку CSRSS загружается единожды для каждого сеанса, ее объекты ALPC-портов можно найти в соответствующих каталогах \Sessions*X*\Windows, как показано на следующем рисунке.



ALPC-сообщение может попасть в одну из четырех различных очередей, реализуемых объектом ALPC-порта:

- **Главная очередь** (main queue). Сообщение было отправлено, и клиент его обрабатывает.
- **Очередь ожидания** (pending queue). Сообщение было отправлено, и вызывающий код ждет ответа, но ответ еще не был отправлен.
- **Большая очередь сообщений** (large message queue). Сообщение было отправлено, но буфер вызывающего кода был слишком мал для его получения. Вызывающему коду дается еще один шанс для выделения более объемного буфера и еще раз запросить полезную нагрузку сообщения.
- **Очередь аннулированных сообщений** (canceled queue). Сообщение, отправленное в адрес порта, было уже отменено.

Обратите внимание на то, что пятая очередь, названная очередью ожидания, не связывает сообщения вместе, вместо этого она связывает все потоки, ожидающие сообщение.

Асинхронные операции

Синхронная модель ALPC связана с исходной LPC-архитектурой в ранней конструкции NT и похожа на другие блокирующие IPC-механизмы, например на Mach-порты. Хотя блокирующий IPC-алгоритм прост по конструкции, в нем имеется множество возможностей для взаимных блокировок, и обходы подобных сценариев приводят к созданию сложного кода, требующего поддержки более гибкой асинхронной (не блокирующей) модели. В качестве таковой ALPC была первоначально разработана для поддержки также и асинхронных операций, которые требовались для масштабируемых RPC-вызовов и других использований, например для поддержки отложенных операций ввода-вывода в драйверах пользовательского режима. Основным свойством ALPC, которое изначально отсутствовало в LPC, является наличие у блокирующих вызовов параметра срока истечения. Это позволяет устаревшим приложениям избегать тех или иных сценариев взаимной блокировки.

Но ALPC-механизм оптимизирован для асинхронных сообщений и предоставляет три различные модели для асинхронных уведомлений. Первая из них на самом деле не уведомляет клиента или сервер, а просто копирует данные полезной нагрузки. Согласно этой модели выбор подходящего метода синхронизации возлагается на того, кто ее реализует. Например, клиент и сервер могут совместно использовать объект события уведомления, или клиент может производить опрос о поступлении данных. Структура данных, используемая этой моделью, — *список завершения* ALPC (который не нужно путать с имеющимся в Windows портом завершения ввода-вывода). Список завершения ALPC является эффективной, неблокирующей структурой, допускающей атомарную передачу данных между клиентами и ее внутренними составляющими, рассматриваемыми далее в разделе «Производительность».

Следующая модель уведомления является ожидающей моделью, использующей имеющийся в Windows механизм порта завершения (вдобавок к списку завершения ALPC). Это позволяет потоку извлекать сразу несколько полезных нагрузок, контролировать максимальное количество конкурирующих запросов и получать преимущество от функциональных возможностей, присущих порту завершения. Реализация пула потоков пользовательского режима предоставляет внутренние API-функции, которые используются процессами для управления ALPC-сообщениями в рамках одной инфраструктуры, такой как рабочие потоки, которая реализуется с использованием этой модели. RPC-система в Windows также использует эту функциональность при использовании локального RPC-вызова (через ncalrpc) для предоставления эффективной доставки сообщения, пользуясь этой поддержкой ядра.

И наконец, поскольку драйверы могут также использовать асинхронные ALPC, но обычно не поддерживают порты завершения на таком высоком уровне, ALPC также предоставляет механизм для более основательного оповещения, основанного на использовании ядра с использованием объектов обратного вызова исполняющей системы. Драйвер может зарегистрировать свою собственную функцию обратного вызова и контекст с помощью функции NtSetInformationAlpcPort, после чего она будет вызвана при получении сообщения. Например, имеющиеся в ядре интерфейсы пользовательского режима диспетчера электропитания используют этот механизм для асинхронных операций подсветки жидкокристаллических дисплеев.

Просмотры, области и разделы

Вместо пересылки буферов сообщений между своими двумя соответствующими процессами, сервер и клиент могут выбрать более эффективный механизм передачи данных, который находится в основе диспетчера памяти Windows: объект раздела. Это позволяет выделить часть памяти для совместного использования и открыть как для клиента, так и для сервера последовательный и одинаковый просмотр этой памяти. По этому сценарию может передаваться столько данных, сколько может поместиться в эту память, и данные просто копируются в один диапазон адресов и тут же становятся доступны в другом диапазоне адресов. К сожалению, связь с использованием общей памяти, такая, которую традиционно предоставляет LPC-вызов, имеет свою долю недостатков, особенно если брать в расчет вопросы безопасности. Например, поскольку как клиент, так и сервер должны иметь доступ к совместно используемой памяти, непривилегированный клиент может воспользоваться этим для повреждения серверной общей памяти и даже для создания выполняемой полезной нагрузки с потенциально вредоносным кодом. Кроме того, поскольку клиент знает, где находятся серверные данные, он может воспользоваться этой информацией для обхода ASLR-защиты.

ALPC предоставляет свою собственную защиту вдобавок к той, которая предоставляется объектами раздела. При использовании ALPC указанный объект раздела ALPC должен быть создан с помощью соответствующей API-функции NtAlpcCreatePortSection, которая создаст корректные ссылки на порт, а также позволит применить автоматический сбор мусора в разделе (существуют также API-функции для ручного удаления мусора). Как только владелец объекта раздела ALPC приступает к его использованию, из выделенных участков памяти составляются области ALPC, которые представляют диапазон используемых внутренних адресов, и добавляют дополнительную ссылку на сообщение. И наконец, внутри диапазона совместно используемой памяти клиенты получают возможности просмотра этой памяти, представляющие собой локальные отображения внутри их адресного пространства.

Области также поддерживают несколько вариантов обеспечения безопасности. Прежде всего области могут быть отображены либо с использование режима безопасности, либо в небезопасном режиме. При использовании режима безопасности для раздела допускаются только два просмотра (отображения). Это обычно используется в том случае, когда серверу нужно иметь совместно используемые данные, имеющие закрытый характер, только с одним клиентским процессом. Кроме того, только одна область для заданного диапазона совместно используемой памяти может быть открыта в рамках применения данного порта. И наконец, области могут также быть помечены защищенными от любого доступа, кроме как доступа по чтению, что разрешает только одному контексту процесса (серверу) иметь доступ по записи к просматриваемой области (это делается с использованием функции MmSecureVirtualMemoryAgainstWrites). Тем временем все остальные клиенты будут иметь доступ только по чтению. Такие настройки подавляют многие атаки, связанные с повышением уровня привилегий, которые могут произойти во время атак на совместно используемую память, и они делают ALPC-вызовы более устойчивыми по сравнению с обычными IPC-механизмами.

Атрибуты

Механизм ALPC предоставляет не только простую передачу сообщений, он также позволяет определенной контекстной информации быть добавленной к каждому сообщению и допускает со стороны ядра отслеживание достоверности, срока службы и осуществление реализации этой информации. Пользователи ALPC имеют также возможность назначения своей собственной пользовательской контекстной информации. Как при системном, так и при пользовательском управлении в ALPC эти данные называются атрибутами. Ядро управляет тремя из них:

- атрибутом безопасности, в котором содержится ключевая информация, позволяющая заимствовать права клиента, а также получать расширенные функциональные возможности безопасности ALPC (которые будут рассмотрены чуть позже);
- атрибутами просмотра данных, отвечающими за управление различными просмотрами, которые связаны с областями ALPC-раздела;
- атрибутом дескрипторов, который содержит информацию о том, какие дескрипторы нужно связывать с сообщением (более подробное описание см. далее в разделе «Безопасность»).

Обычно эти атрибуты изначально передаются сервером или клиентом при отправке сообщения и превращаются в принадлежащее ядру внутреннее ALPC-представление. Если пользователь ALPC запрашивает эти данные назад, они безопасно выставляются обратно. При реализации такой модели и ее совмещении с его собственной внутренней таблицей дескрипторов, которая будет рассмотрена далее, механизм ALPC может непрозрачно содержать важные данные между клиентами и серверами, сохраняя при этом истинные указатели в режиме ядра.

И наконец, поддерживается четвертый атрибут, который называется атрибутом контекста. Этот атрибут поддерживает традиционный LPC-стиль, характерный для пользователя указатель контекста, который может быть связан с данным сообщением, и он по-прежнему поддерживается для сценариев, где с парой клиент-сервер должны быть связаны какие-нибудь пользовательские данные.

Для правильного определения атрибутов внутренние ALPC-потребители располагают различными API-функциями, среди которых AlpcInitializeMessageAttribute и AlpcGetMessageAttribute.

Блобы, дескрипторы и ресурсы

Хотя библиотека ALPC показывает только один тип объекта диспетчера объектов (порт), внутри системы она должна управлять множеством структур данных, что позволяет ей выполнять задачи, требуемые ее механизмам. Например, ALPC нуждается в размещении и отслеживании сообщений, связанных с каждым портом, а также атрибутов этих сообщений, которые она должна отслеживать на всем протяжении срока их существования. Вместо использования для управления данными процедур диспетчера объектов, ALPC реализует свои собственные облегченные объекты, называемые *блобами* (blobs). Как и объекты, блобы могут быть автоматически размещены в памяти и удалены из нее при сборке мусора, отслежены по ссылкам и заблокированы через механизмы синхронизации. Кроме

этого, блобы могут иметь специализированные функции обратного вызова, занимающиеся их размещением и освобождением занимаемой ими памяти, которые позволяют их владельцам управлять дополнительной информацией, которая может понадобиться для отслеживания каждого блоба. И наконец, ALPC также использует имеющуюся в исполняющей системе реализацию таблицы дескрипторов (она необходима для объектов и идентификаторов процессов и идентификатором потоков) для того, чтобы получить таблицу дескрипторов, специализированную под ALPC, которая позволяет ALPC вместо использования указателей генерировать для блобов закрытые дескрипторы.

Блобами в модели ALPC являются, к примеру, сообщения, и их конструктор генерирует идентификатор сообщения, который сам по себе является дескриптором в таблице дескрипторов ALPC. В число других блобов ALPC входят:

- блоб подключения, в котором хранятся порты связи клиента и сервера, а также порт подключения к серверу и таблица дескрипторов ALPC;
- блоб безопасности, в котором хранятся данные безопасности, необходимые для заимствования прав клиента. В нем хранится атрибут безопасности;
- блобы раздела, области и просмотра, которые описывают используемую в ALPC модель совместного использования памяти. Блоб просмотра в конечном итоге отвечает за хранение атрибута просмотра данных;
- резервный блоб, реализующий поддержку для резервных объектов ALPC (см. раздел «Резервные объекты»);
- блоб дескриптора данных, в котором содержится информация, позволяющая поддерживать атрибут дескрипторов ALPC.

Поскольку блобы выделяются из выгружаемой памяти, они должны тщательно отслеживаться для своевременного удаления. Для некоторых разновидностей блобов это не составляет труда: например, когда освобождается сообщение ALPC, то вместе с ним удаляется и содержавший его блоб. Но некоторые блобы могут представлять сразу несколько атрибутов, прикрепленных к одному ALPC-сообщению, и ядро должно соответствующим образом управлять сроком их существования. Например, поскольку у сообщения может быть несколько связанных с ним просмотров (когда доступ к одной и той же совместно используемой памяти есть сразу у нескольких клиентов), просмотры должны отслеживаться наряду со ссылающимися на них сообщениями. ALPC использует концепцию ресурсов для реализации этой возможности. Каждое сообщение связано со списком ресурсов, и при размещении блоба, связанного с сообщением (которое не является простым указателем), он также добавляется в качестве ресурса сообщения. В свою очередь, библиотека ALPC предоставляет функциональную возможность для поиска, смещения и удаления связанных ресурсов. Блобы безопасности, резервные блобы и блобы просмотра хранятся в качестве ресурсов.

Безопасность

ALPC реализует несколько механизмов безопасности, полноценные границы безопасности и смягчение условий для предотвращения атак в случае общих ошибок разбора IPC. На базовом уровне объекты портов ALPC управляются

теми же интерфейсами диспетчера объектов, который управляет объектами в безопасном режиме, препятствуя непривилегированным приложениям в получении дескрипторов на порты сервера с ACL. Кроме того, ALPC предоставляет модель доверия на основе идентификатора безопасности (SID), унаследованную у исходной конструкции LPC. Эта модель позволяет клиентам проверять достоверность сервера, к которому они подключаются, основываясь на большем, чем просто на имени порта. С использованием безопасного порта клиентский процесс отправляет ядру SID ожидаемого им на стороне конечной точки серверного процесса. Во время подключения ядро проверяет достоверность того, что клиент действительно подключился к ожидаемому серверу, снижая тем самым возможность атаки с незаконным проникновением в пространство имен, когда непроверенный сервер создает порт для имитации настоящего сервера.

ALPC также позволяет как клиентам, так и серверам в рамках атомарной операции однозначно идентифицировать поток и процесс, ответственный за каждое сообщение. Этот механизм также полностью поддерживает имеющуюся в Windows модель заимствования прав, используя для этого API-функцию NtAlpcImpersonateClientThread. Другие API-функции дают серверу ALPC возможность запросить идентификаторы безопасности (SID), связанные со всеми подключенными клиентами, и запросить локальный уникальный идентификатор (LUID, locally unique identifier) маркера безопасности клиента (см. главу 6).

Производительность

ALPC для повышения производительности использует несколько стратегий, главным образом, путем поддержки списков завершения. На уровне ядра список завершения, по сути, является пользовательской таблицей описания памяти — MDL, которая была опробирована и заблокирована, а затем сопоставлена с адресом. Поскольку этот механизм связан с MDL-таблицей (отслеживающей физические страницы), когда клиент отправляет сообщение на сервер, копирование полезной нагрузки может осуществляться непосредственно на физическом уровне, вместо запроса у ядра двойного буферирования сообщения, как это принято в других IPC-механизмах.

Сам список завершения реализован в виде 64-разрядной очереди записей о завершениях, и для вставки и удаления записей из очереди ее потребители могут воспользоваться заблокированной операцией сравнения-обмена, как в пользовательском режиме, так и в режиме ядра. Кроме того, чтобы упростить размещение, сразу после инициализации MDL используется битовая карта для идентификации доступных областей памяти, которые могут использоваться для хранения новых сообщений, которые еще находятся в очереди. Алгоритм битовой карты также использует простые инструкции блокировки процессора для атомарного выделения и освобождения областей физической памяти, которые могут использоваться списками завершения.

Еще одной оптимизацией производительности ALPC является использование *зон сообщений*. Зона сообщений — это просто заранее выделенный буфер ядра (также поддерживаемый MDL), в котором сообщение может быть сохранено до тех пор, пока его не извлечет сервер или клиент. Зона сообщений связывает

системный адрес с сообщением, позволяя ему сделаться видимым в адресном пространстве любого процесса. Более важно в случае использования асинхронных операций то, что при этом не требуется сложных настроек отложенных полезных нагрузок, поскольку неважно, когда именно потребитель, в конечном счете, извлекает данные сообщения, зона сообщений не утратит своей достоверности. И списки завершения, и зоны сообщений могут быть установлены с помощью функции NtAlpcSetInformationPort.

И заключительная оптимизация, о которой стоит упомянуть, заключается в том, что вместо копирования данных сразу же после их отправки, ядро устанавливает полезную нагрузку для отложенной копии, забирает только нужную информацию, но без какого-либо копирования. Данные сообщения копируются только тогда, когда получатель востребует это сообщение. Вполне очевидно, что при использовании зоны сообщений или совместно используемой памяти этот метод не дает никакого преимущества, но при передаче асинхронных, буферируемых в ядре сообщений его можно использовать для оптимизации отмен и для работы сценариев с напряженным трафиком.

Отладка и трассировка

ALPC-сообщения могут быть зарегистрированы на проверенных версиях ядра. Все ALPC-атрибуты, блобы, зоны сообщений и транзакции диспетчера могут быть зарегистрированы в индивидуальном порядке, и регистрационные записи могут быть выведены в виде дампа с помощью отсутствующих в документации команд !alpc. На системах, продаваемых в розницу, IT-администраторы и наладчики могут для отслеживания ALPC-сообщений включить ALPC-регистратор Event Tracing for Windows (ETW). ETW-события не включают данные полезной нагрузки, но в них содержится информация о подключениях, отключениях, а также об отправке-получении и об ожидании-разблокировании. И наконец, даже на розничных системах определенные команды !alpc получают информацию о ALPC-портах и сообщениях.

ЭКСПЕРИМЕНТ: ВЫВОД ДАМПА ПОРТА ПОДКЛЮЧЕНИЯ

В данном эксперименте будет использоваться CSRSS API-порт для Windows-процессов, запущенных в сеансе 1 (Session 1), который обычно является интерактивным сеансом для пользователя консоли. При каждом запуске Windows-приложения оно подключается к API-порту CSRSS в соответствующем сеансе.

1. Начните с получения указателя на порт подключения с помощью команды !object:

```
0: kd> !object \Sessions\1\Windows\ApiPort
Object: fffffa8004dc2090  Type: (fffffa80027a2ed0) ALPC Port
    ObjectHeader: fffffa8004dc2060 (new version)
    HandleCount: 1  PointerCount: 50
    Directory Object: fffff8a001a5fb30  Name: ApiPort
```

2. Теперь выведите информационный дамп самого объекта порта с помощью команды !alpc /p. Это, к примеру, подтвердит, что владельцем является CSRSS:

```
0: kd> !alpc /p fffffa8004dc2090
Port @ fffffa8004dc2090
  Type                      : ALPC_CONNECTION_PORT
  CommunicationInfo         : fffff8a001a22560
    ConnectionPort          : fffffa8004dc2090
    ClientCommunicationPort : 0000000000000000
    ServerCommunicationPort : 0000000000000000
  OwnerProcess              : fffffa800502db30 (csrss.exe)
  SequenceNo                : 0x000003C9 (969)
  CompletionPort            : 0000000000000000
  CompletionList            : 0000000000000000
  MessageZone               : 0000000000000000
  ConnectionPending         : No
  ConnectionRefused         : No
  Disconnected              : No
  Closed                    : No
  FlushOnClose              : Yes
  ReturnExtendedInfo        : No
  Waitable                  : No
  Security                  : Static
  Wow64CompletionList       : No
  Main queue is empty.
  Large message queue is empty.
  Pending queue is empty.
  Canceled queue is empty.
```

3. Можно увидеть, какие клиенты подключены к порту, к числу которых будут относиться все Windows-процессы, запущенные в этом сеансе, для чего нужно воспользоваться отсутствующей в документации командой !alpc /lpc. Кроме того, будут видны порты связи сервера и клиента, относящиеся к каждому подключению, и любые ожидающие сообщения в любой из очередей:

```
0: kd> !alpc /lpc fffffa8004dc2090
Port @fffffa8004dc2090 has 14 connections
SRV:fffffa8004809c50 (m:0, p:0, l:0) <-> CLI:fffffa8004809e60 (m:0, p:0, l:0),
 Process=fffffa8004ffcb30 ('winlogon.exe')
SRV:fffffa80054dfb30 (m:0, p:0, l:0) <-> CLI:fffffa80054dfe60 (m:0, p:0, l:0),
 Process=fffffa80054de060 ('dwm.exe')
SRV:fffffa8005394dd0 (m:0, p:0, l:0) <-> CLI:fffffa80054e1440 (m:0, p:0, l:0),
 Process=fffffa80054e2290 ('winvnc.exe')
SRV:fffffa80053965d0 (m:0, p:0, l:0) <-> CLI:fffffa8005396900 (m:0, p:0, l:0),
 Process=fffffa80054ed060 ('explorer.exe')
SRV:fffffa80045a8070 (m:0, p:0, l:0) <-> CLI:fffffa80045af070 (m:0, p:0, l:0),
 Process=fffffa80045b1340 ('logonhlp.exe')
SRV:fffffa8005197940 (m:0, p:0, l:0) <-> CLI:fffffa800519a900 (m:0, p:0, l:0),
 Process=fffffa80045da060 ('TSVNCache.exe')
SRV:fffffa800470b070 (m:0, p:0, l:0) <-> CLI:fffffa800470f330 (m:0, p:0, l:0),
 Process=fffffa8004713060 ('vmware-tray.ex')
SRV:fffffa80045d7670 (m:0, p:0, l:0) <-> CLI:fffffa80054b16f0 (m:0, p:0, l:0),
 Process=fffffa80056b8b30 ('WINWORD.EXE')
```

продолжение ⇩

```
SRV:fffffa80050e0e60 (m:0, p:0, l:0) <-> CLI:fffffa80056fee60 (m:0, p:0, l:0),
 Process=fffffa800478f060 ('Winobj.exe')
SRV:fffffa800482e670 (m:0, p:0, l:0) <-> CLI:fffffa80047b7680 (m:0, p:0, l:0),
 Process=fffffa80056aab30 ('cmd.exe')
SRV:fffffa8005166e60 (m:0, p:0, l:0) <-> CLI:fffffa80051481e0 (m:0, p:0, l:0),
 Process=fffffa8002823b30 ('conhost.exe')
SRV:fffffa80054a2070 (m:0, p:0, l:0) <-> CLI:fffffa80056e6210 (m:0, p:0, l:0),
 Process=fffffa80055669e0 ('livekd.exe')
SRV:fffffa80056aa390 (m:0, p:0, l:0) <-> CLI:fffffa80055a6c00 (m:0, p:0, l:0),
 Process=fffffa80051b28b0 ('livekd64.exe')
SRV:fffffa8005551d90 (m:0, p:0, l:0) <-> CLI:fffffa80055bfc60 (m:0, p:0, l:0),
 Process=fffffa8002a69b30 ('kd.exe')
```

4. Учтите, что при наличии других сеансов этот эксперимент можно повторить и для них (а также для сеанса 0, то есть для системного сеанса). В конечном счете, будет получен список всех Windows-процессов на вашей машине. Если используется подсистема для приложений UNIX, эту технологию можно также применить для объекта \PSXSS\ApiPort. ■

Отслеживание событий ядра

Различные компоненты ядра Windows и некоторые основные драйверы устройств приспособлены для записи данных отслеживания их операций с целью использования этих данных при поиске причин системных сбоев. Эта возможность основана на общей инфраструктуре ядра, которая предоставляет данные отслеживания средству отслеживания событий ETW, работающему в пользовательском режиме. Приложения, использующие ETW, можно отнести к одной или к нескольким из трех категорий:

- **Контроллер.** Контроллер запускает и останавливает сеансы регистрации и управляет пулами буферов. В качестве примера можно привести Монитор надежности и производительности (см. врезку «Эксперимент: отслеживание активности TCP/IP с помощью регистратора ядра», с. 268) и средство XPerf из набора Windows Performance Toolkit (см. врезку «Эксперимент: отслеживание активности прерываний и DPC» в этой главе, с. 135).
- **Поставщик.** Поставщик определяет глобальные уникальные идентификаторы — GUID (globally unique identifiers) — для классов событий, данные об отслеживании которых он может предоставить, и регистрирует их с помощью ETW. Поставщик принимает команды от контроллера для запуска и остановки отслеживания классов событий, за которые он отвечает.
- **Потребитель.** Потребитель выбирает один или несколько сеансов отслеживания, из которых ему нужно считывать данные. Потребители могут получать события в буферах в реальном масштабе времени или в регистрационных файлах.

Windows содержит в себе десятки поставщиков пользовательского режима для всего, начиная с таких средств, как Active Directory и диспетчер управления службами (Service Control Manager), и заканчивая таким средством, как Explorer. В ETW также определяется сеанс регистрации NT Kernel Logger (также извест-

ный, как регистратор ядра) для использования ядром и основными драйверами. Поставщики для NT Kernel Logger реализуются кодом ETW в файле Ntoskrnl.exe и основными драйверами.

Когда контроллер в пользовательском режиме включает регистратор ядра, ETW-библиотека (реализованная в \Windows\System32\Ntdll.dll) вызывает системную функцию NtTraceControl, сообщает ETW-коду в ядре, к отслеживанию каких классов событий нужно приступить. Если сконфигурирован регистрационный файл (в отличие от регистрации в буфере внутри памяти), ядро создает системный поток в системном процессе, который создает файл журнала регистрации. Когда ядро получает отслеживаемые события из разрешенных источников отслеживания, оно записывает их в буфер. Будучи запущенным, поток регистрационного файла активизируется один раз в секунду, чтобы сбросить содержимое буферов в файл журнала.

Записи отслеживания, сгенерированные регистратором ядра, имеют стандартный ETW-заголовок отслеженного события, в который записывается отметка времени, идентификаторы процесса и потока, а также информация о том, к какому классу событий относится запись. Классы событий могут предоставлять дополнительные данные, характерные для своих событий. Например, записи отслеженного события, относящегося к классу событий диска, показывают тип операции (чтение или запись), номер диска, на который направлена операция, а также смещение сектора и длину операции.

Вот некоторые классы, отслеживание которых может быть включено в регистраторе ядра, и компоненты, генерирующие каждый класс:

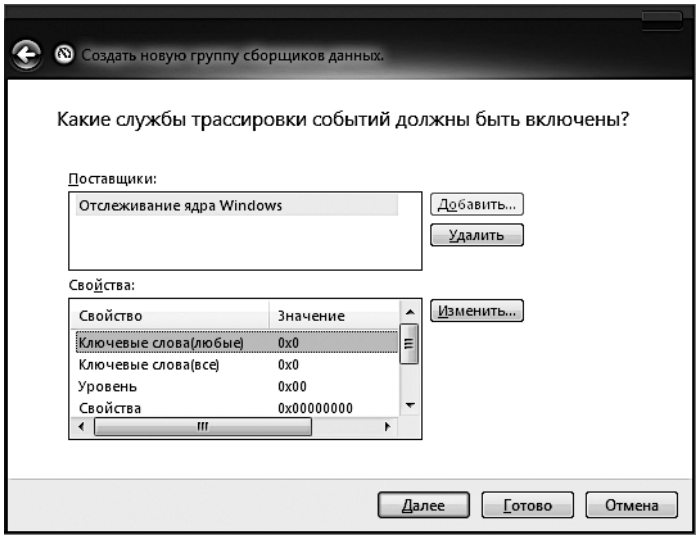
- **Disk I/O** (Дисковый ввод-вывод). Драйвер класса дисков.
- **File I/O** (Файловый ввод-вывод). Драйверы файловой системы.
- **File I/O Completion** (Завершение файлового ввода-вывода). Драйверы файловой системы.
- **Hardware Configuration** (Настройка оборудования). Диспетчер устройств plug and play.
- **Image Load/Unload** (Загрузка-выгрузка образов). Системный загрузчик образов в ядре.
- **Page Faults** (Ошибки обращения к странице). Диспетчер памяти.
- **Hard Page Faults** (Ошибки обращения к странице, требующие ее считывания с диска). Диспетчер памяти.
- **Process Create/Delete** (Создание-удаление процесса). Диспетчер процессов (см. главу 5).
- **Thread Create/Delete** (Создание-удаление потока). Диспетчер процессов.
- **Registry Activity** (Активность системного реестра). Диспетчер конфигурации (см. главу 4).
- **Network TCP/IP** (Сеть TCP/IP). Драйвер TCP/IP.
- **Process Counters** (Счетчики процесса). Диспетчер процессов.
- **Context Switches** (Переключения контекста). Диспетчер ядра.
- **Deferred Procedure Calls** (Вызовы отложенных процедур). Диспетчер ядра.
- **Interrupts** (Прерывания). Диспетчер ядра.
- **System Calls** (Системные вызовы). Диспетчер ядра.

- **Sample Based Profiling** (Профилирование на основе образцов). Диспетчер ядра и HAL.
- **Driver Delays** (Задержки драйвера). Диспетчер ввода-вывода.
- **Split I/O** (Раздробленный ввод-вывод). Диспетчер ввода-вывода.
- **Power Events** (События электропитания). Диспетчер электропитания.
- **ALPC.** Усовершенствованный вызов локальной процедуры.
- **Scheduler and Synchronization** (События планировщика и синхронизации). Диспетчер ядра (см главу 5).

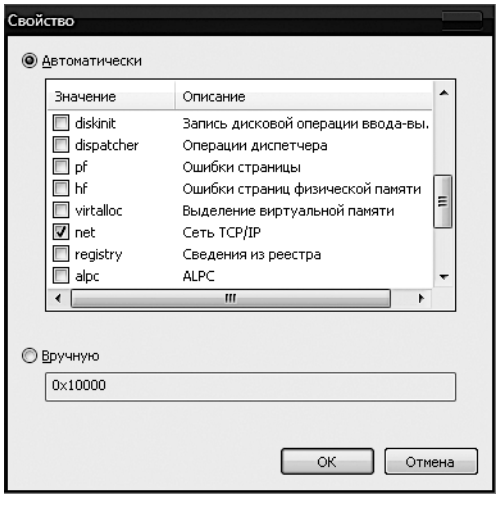
ЭКСПЕРИМЕНТ: ОТСЛЕЖИВАНИЕ АКТИВНОСТИ TCP/IP С ПОМОЩЬЮ РЕГИСТРАТОРА ЯДРА

Чтобы включить регистратор ядра и заставить его сгенерировать файл журнала активности TCP/IP, выполните следующие действия:

1. Запустите Системный монитор (Performance Monitor) и щелкните на пунктах Группы сборщиков данных (Data Collector Sets) и Особый (User Defined).
2. Щелкните правой кнопкой мыши на пункте Особый (User Defined), выберите пункт Создать (New), а затем пункт Группа сборщиков данных (Data Collector Set).
3. При появлении приглашения на ввод введите имя для группы сборщиков данных (например, «эксперимент»), и перед тем как щелкнуть на кнопке Далее (Next), выберите пункт Создать вручную (для опытных) (Create Manually (Advanced)).
4. В открывшемся диалоговом окне выберите пункт Создать журналы данных (Create Data Logs), установите флажок Данные отслеживания событий (Event Trace Data), а затем щелкните на кнопке Далее (Next). В области Поставщики (Providers) щелкните на кнопке Добавить (Add) и найдите пункт Отслеживание ядра Windows (Windows Kernel Trace). В списке Свойства (Properties) выберите пункт Ключевые слова (любые) (Keywords(Any)), а затем щелкните на кнопке Изменить (Edit).



5. Из этого списка выберите только Net для Сеть TCP/IP (Network TCP/IP), а затем щелкните на кнопке OK.



6. Щелкните на кнопке Далее (Next), чтобы выбрать место хранения файлов. По умолчанию этим местом будет C:\Perflogs\<Пользователь>\эксперимент\, то есть оно будет соответствовать имени, которое вы дали группе сборщиков данных. Щелкните на кнопке Далее (Next) и в редактируемом поле Запуск от имени (Run As) введите учетное имя администратора и укажите соответствующий ему пароль. Щелкните на кнопке Готово (Finish). Теперь вы должны увидеть окно, похожее на это.

7. Щелкните правой кнопкой на пункте «эксперимент» (или на том имени, которое вы дали группе сборщиков данных), а затем щелкните на пункте Пуск (Start). Теперь проведите несколько сетевых действий, открыв браузер и посетив веб-сайт.
8. Щелкните еще раз правой кнопкой мыши на группе сборщиков данных, а затем щелкните на пункте Стоп (Stop).
9. Откройте окно командной строки и перейдите в каталог C:\Perflogs\эксперимент\00001 (or the directory into which you specified that the trace log file be stored).
10. Запустите команду tracerpt, передава ей имя файла журнала отслеживания:

```
tracerpt DataCollector01.etl –o dumpfile.csv –of CSV
```

11. Откройте файл dumpfile.csv в Microsoft Excel или в текстовом редакторе. Вы увидите записи отслеживания TCP и (или) UDP.

Tcpip	SendIPV4	0xFFFFFFFF	1.28663E+17	0	0	1992	1388	157.54.86.28	172.31.234.35	80	49414	646659	646661
Udpip	RecvIPV4	0xFFFFFFFF	1.28663E+17	0	0	4	50	172.31.239.255	172.31.233.110	137	137	0	0x0
Udpip	RecvIPV4	0xFFFFFFFF	1.28663E+17	0	0	4	50	172.31.239.255	172.31.234.162	137	137	0	0x0
Tcpip	RecvIPV4	0xFFFFFFFF	1.28663E+17	0	0	1992	1425	157.54.86.28	172.31.234.35	80	49414	0	0x0
Tcpip	RecvIPV4	0xFFFFFFFF	1.28663E+17	0	0	1992	1380	157.54.86.28	172.31.234.35	80	49414	0	0x0
Tcpip	RecvIPV4	0xFFFFFFFF	1.28663E+17	0	0	1992	45	157.54.86.28	172.31.234.35	80	49414	0	0x0
Tcpip	RecvIPV4	0xFFFFFFFF	1.28663E+17	0	0	1992	1415	157.54.86.28	172.31.234.35	80	49414	0	0x0
Tcpip	RecvIPV4	0xFFFFFFFF	1.28663E+17	0	0	1992	740	157.54.86.28	172.31.234.35	80	49414	0	0x0

Дополнительную информацию о ETW и о регистраторе ядра, включая образцы кода для контроллеров и потребителей, можно найти в Windows SDK.

Wow64

Wow64 (Эмуляция Win32 на 64-разрядной версии Windows) относится к программному обеспечению, которое позволяет выполнять 32-разрядные x86-приложения на 64-разрядной версии Windows. Этот механизм реализован в виде набора DLL-библиотек пользовательского режима, с определенной поддержкой от ядра для создания 32-разрядных версий того, что в обычных условиях было бы только 64-разрядными структурами данных, например блока окружения процесса (process environment block, PEB) и блока окружения потока (thread environment block, TEB).

Ядром также реализуется изменение контекста Wow64 с помощью функций Get/SetThreadContext. За работу Wow64 отвечают следующие DLL-библиотеки пользовательского режима:

- **Wow64.dll.** Управляет созданием процесса и потока и отлавливает основные системные вызовы и системные вызовы, связанные с исключениями и диспетчеризацией, экспортируемые Ntoskrnl.exe. В ней также реализуются перенаправления, связанные с файловой системой и с реестром.
- **Wow64Cpu.dll.** Управляет 32-разрядным контекстом центрального процессора каждого потока, запущенного в рамках Wow64, и предоставляет характерную для архитектуры того или иного процессора поддержку для переключения режима его работы с 32-разрядного в 64-разрядный, и наоборот.
- **Wow64Win.dll.** Перехватывает все системные вызовы GUI, экспортируемые Win32k.sys.
- **IA32Exec.bin и Wowia32x.dll на системах IA64.** Содержит программный эмулятор IA-32 и библиотеку его интерфейса. Поскольку процессоры Itanium в силу своей конструкции не могут эффективно выполнять 32-разрядные инструкции x86 (производительность снижается более чем на 30 %), требуется программная эмуляция (через двоичную трансляцию) с использованием этих двух дополнительных компонентов.

Взаимоотношения между этими DLL-библиотеками показаны на рис. 3.31.

Схема адресного пространства процессов Wow64

Процессы Wow64 могут запускаться с 2 Гбайт или 4 Гбайт виртуальным адресным пространством. Если в заголовке образа установлен флаг оповещения о не-

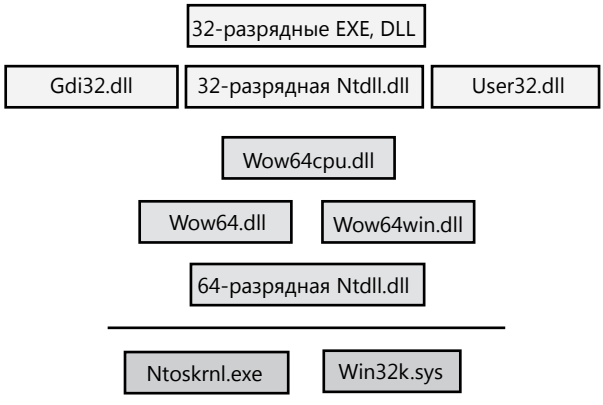


Рис. 3.31. Архитектура Wow64

обходимости расширенного адресного пространства (large-address-aware flag), диспетчер памяти резервирует адресное пространство пользовательского режима, выходящее за 4 Гбайт границу до конца границы пространства пользовательского режима. Если образ не помечен, как оповещенный о необходимости расширенного адресного пространства, диспетчер памяти резервирует адресное пространство пользовательского режима, превышающее 2 Гбайт.

Системные вызовы

Wow64 отлавливает те пути выполнения кода, где 32-разрядный код должен переходить к исходной 64-разрядной системе или когда исходная система нуждается в вызове 32-разрядного кода пользовательского режима. В ходе создания процесса диспетчер процессов отображает на адресное пространство процесса исходную 64-разрядную библиотеку Ntdll.dll, а также 32-разрядную библиотеку Ntdll.dll для процессов Wow64. Когда вызывается инициализация загрузчика, она вызывает код инициализации Wow64, находящийся внутри библиотеки Wow64.dll. Затем Wow64 устанавливает контекст запуска, нужный 32-разрядной библиотеке Ntdll, переключает режим работы центрального процессора на 32-разрядный и приступает к выполнению кода 32-разрядного загрузчика. С этого момента выполнение продолжается таким образом, будто процесс запущен на настоящей 32-разрядной системе.

Специальные 32-разрядные версии библиотек Ntdll.dll, User32.dll и Gdi32.dll находятся в папке \Windows\Syswow64 (наряду с некоторыми другими DLL-библиотеками, осуществляющими связь между процессами, например Rpcrt4.dll). Вместо запуска реальных 32-разрядных инструкций системного вызова они осуществляют вызовы в механизм Wow64, а тот, в свою очередь, осуществляет переход в исходный 64-разрядный режим, захватывая параметры, связанные с системным вызовом (преобразуя 32-разрядные указатели в 64-разрядные), и выдает соответствующий, характерный для структуры 64-разрядный системный вызов. Когда осуществляется возврат из системного вызова, Wow64 перед возвращением в 32-разядный режим преобразует любые выходные параметры, если это необходимо, из 64-разрядного в 32-разрядный формат.

Диспетчеризация исключений

Wow64 захватывает диспетчеризацию исключений с помощью имеющейся в Ntdll-библиотеке функции KiUserExceptionDispatcher. Когда 64-разрядное ядро собирается провести диспетчеризацию исключения Wow64-процесса, Wow64 захватывает исходное исключение и запись контекста в пользовательском режиме, а затем подготавливает 32-разрядное исключение и запись контекста и проводит его диспетчеризацию, как будто она осуществляется в настоящем 32-разрядном ядре.

Диспетчеризация пользовательских APC

Wow64 также захватывает доставку APC пользовательского режима с помощью имеющейся в Ntdll-библиотеке функции KiUserApcDispatcher. Когда 64-разрядное ядро собирается провести диспетчеризацию APC пользовательского режима Wow64-процесса, оно отображает 32-разрядный APC-адрес на более высокий диапазон 64-разрядного адресного пространства. Затем 64-разрядная Ntdll-библиотека захватывает исходный APC-вызов и запись контекста в пользовательском режиме и отображает его обратно на 32-разрядный адрес. Затем она подготавливает 32-разрядный APC-вызов и запись контекста и проводит диспетчеризацию точно так же, как это делается в настоящем 32-разрядном ядре.

Поддержка консоли

Поскольку поддержка консоли реализована в пользовательском режиме с помощью файла Csrss.exe, в котором присутствует только двоичный код, характерный для системы, 32-разрядные приложения не смогут выполнять консольный ввод-вывод при выполнении на 64-разрядной версии Windows. Точно так же, как для преобразования 32-разрядных RPC-вызовов в 64-разрядные существует специальная библиотека rpcrt4.dll, в 32-разрядной библиотеке Kernel.dll для Wow64 содержится специальный код для вызова в рамках Wow, предназначенный для преобразования параметров во время взаимодействия с Csrss и Conhost.exe.

Пользовательские функции обратного вызова

Wow64 перехватывает все обратные вызовы из ядра в пользовательский режим. Wow64 рассматривает такие вызовы, как системные, но преобразование данных производится в обратном порядке: входные параметры преобразуются из 64 в 32 разряда, а выходные параметры преобразуются, когда обратный вызов возвращается из 32 в 64 разряда.

Перенаправления в файловой системе

Для совместимости приложений и уменьшения усилия по переносу приложений из Win32 в 64-разрядную версию Windows, была сохранена прежняя система названий каталогов. Поэтому в папке \Windows\System32 содержатся исходные 64-разрядные образы. В ходе захвата Wow64 всех системных вызовов эта структура переводит все, связанные с путевыми именами API-функции и заменяет путевое имя папки \Windows\System32 на \Windows\Syswow64. Wow64 также перенаправляет \Windows\LastGood на \Windows\LastGood\syswow64,

и \Windows\Regedit.exe на \Windows\syswow64\Regedit.exe. Благодаря использованию системных переменных среды переменная %PROGRAMFILES% также настроена на папку \Program Files (x86) для 32-разрядных приложений, в то время как для 64-разрядных приложений она настроена на папку \Program Files. Существуют также переменные CommonProgramFiles и CommonProgramFiles (x86), которые всегда указывают на размещение 32-разрядных приложений, в то время как переменные ProgramW6432 и CommonProgramW6432 несомненно указывают на размещение 64-разрядных приложений.

ПРИМЕЧАНИЕ

Поскольку некоторые 32-разрядные приложения действительно могут знать о 64-разрядных образах и работать с ними, виртуальный каталог, \Windows\Sysnative, позволяет вводу-выводу в этот каталог, инициируемому из 32-разрядного приложения, быть исключенным из файлового перенаправления. На самом деле такого каталога нет, это виртуальный путь, допускающий доступ к реальному каталогу System32 даже из приложения, запущенного под управлением Wow64.

Существует ряд подкаталогов \Windows\System32, которые в целях совместимости исключены из числа перенаправляемых, поскольку попытки доступа к ним предпринимаются 32-разрядными приложениями, которые на самом деле обращаются к реальным каталогам. К их числу относятся:

- %windir%\system32\drivers\etc
- %windir%\system32\spool
- %windir%\system32\catroot и %windir%\system32\catroot2
- %windir%\system32\logfiles
- %windir%\system32\driverstore

И наконец, Wow64 предоставляет механизм управления встроенным в Wow64 перенаправлением файловой системы Wow64 на основе отдельных потоков с помощью функций Wow64DisableWow64FsRedirection и Wow64RevertWow64FsRedirection. У этого механизма могут быть проблемы с DLL-библиотеками, загружаемыми с задержкой, с открытием файлов через общее файловое диалоговое окно и даже с локализацией, поскольку при отключении перенаправления система больше его не использует в ходе какой либо внутренней загрузки, и некоторые только 64-разрядные файлы после это могут быть не найдены. Как правило, с точки зрения методологии разработчикам будет безопаснее использовать пути c:\windows\sysnative или некоторые другие представленные ранее постоянные пути.

Перенаправления в реестре

Приложения и компоненты хранят свои настроечные данные в реестре. Компоненты обычно пишут свои настроечные данные в реестр, когда они регистрируются в ходе установки. Если один и тот же компонент установлен и зарегистрирован как 32-разрядный двоичный код и как 64-разрядный двоичный код, последний зарегистрированный компонент перепишет регистрацию предыдущего компонента, поскольку оба они ведут запись в одно и то же место реестра.

Чтобы помочь в явном решении этой проблемы без внесения каких-нибудь кодовых изменений в 32-разрядные компоненты, реестр разбит на две части: исходный и Wow64. По умолчанию 32-разрядные компоненты обращаются к 32-разрядному представлению, а 64-разрядные компоненты обращаются к 64-разрядному представлению. Это обеспечивает безопасную среду выполнения для 32-разрядных и для 64-разрядных компонентов и отделяет состояние 32-разрядных приложений от 64-разрядных приложений, если таковые существуют.

Для реализации такого положения вещей Wow64 перехватывает все системные вызовы, открывающие разделы реестра, и ретранслирует путь к разделу так, чтобы он указывал на Wow64-представление реестра. Wow64 разбивает реестр в следующих точках:

- HKLM\SOFTWARE
- HKEY_CLASSES_ROOT

Но следует заметить, что многие подразделы фактически используются совместно 32-разрядными и 64-разрядными приложениями, то есть разбивается не весь раздел.

В каждом из этих разделов Wow64 создает подраздел под названием Wow6432Node. В этом подразделе хранится 32-разрядная настроечная информация. Все остальная часть реестра используется совместно 32-разрядными и 64-разрядными приложениями (например, HKLM\SYSTEM).

В качестве дополнительной помощи, если 32-разрядное приложение записывает в реестр значение параметра REG_SZ или параметра REG_EXPAND_SZ, начинающееся с данных «%ProgramFiles%» или «%commonprogramfiles%», Wow64 изменяет фактические значения на «%ProgramFiles(x86)%» и «%commonprogramfiles(x86)%», чтобы было соответствие с перенаправлениями в файловой системе и с той схемой, которая была рассмотрена ранее. В данном случае 32-разрядное приложение должно вести запись именно этих строк, любые другие данные будут проигнорированы и записаны обычным способом. И наконец, любой раздел, содержащий строковое значение «system32», во всех случаях заменяется разделом, содержащим «syswow64», независимо от установленных флагов и чувствительности к регистру букв, пока не будет использован флаг KEY_WOW64_64KEY и раздел не будет в списке отраженных разделов — «reflected keys», доступном на MSDN.

Для приложений, нуждающихся в явном указании раздела реестра для определенного представления, это разрешают сделать в функциях RegOpenKeyEx, RegCreateKeyEx, RegOpenKeyTransacted, RegCreateKeyTransacted и RegDeleteKeyEx следующие флаги:

- KEY_WOW64_64KEY — явно указывает на открытие 64-разрядного раздела из 32-разрядного или из 64-разрядного приложения и отключает рассмотренный ранее перехват REG_SZ или REG_EXPAND_SZ;
- KEY_WOW64_32KEY — явно указывает на открытие 32-разрядного раздела из 32-разрядного или из 64-разрядного приложения.

Запросы на управление вводом-выводом

Кроме обычных операций чтения и записи приложения могут вести обмен данными с некоторыми драйверами устройств через функции управления вводом-

выводом устройства, используя Windows API-функцию DeviceIoControl. Вместе с вызовом приложение может указать буфер ввода и (или) буфер вывода. Если буфер содержит данные, зависящие от указателя, и процесс, отправляющий запрос на управление, является Wow64-процессом, представление структуры ввода и (или) структуры вывода у 32-разрядного приложения и 64-разрядного драйвера разные, поскольку указатели для 32-разрядных приложений состоят из 4 байт, а для 64-разрядных приложений из 8 байт. В таком случае ожидается, что драйвер ядра преобразует соответствующие структуры, зависящие от указателей. Для определения, исходит ли запрос на ввод-вывод из процесса Wow64, драйверы могут вызвать функцию IoIs32bitProcess[1].

16-разрядные программы установки

Wow64 не поддерживает выполнение 16-разрядных приложений. Но поскольку многие установщики приложений являются 16-разрядными программами, в Wow64 имеется специальный код для ссылок на работу конкретных, широко известных 16-разрядных установщиков, к числу которых относятся:

- Microsoft ACME Setup версии: 1.2, 2.6, 3.0 и 3.1;
- InstallShield версии 5.x (где x является любым номером промежуточной версии).

При каждой попытке создания с помощью API-функции CreateProcess() 16-разрядного процесса загружается библиотека Ntvdm64.dll, которой передается управление для проверки, не является ли 16-разрядный исполняемый код одним из поддерживаемых установщиков. Если это так и есть, для запуска 32-разрядной версии установщика с теми же аргументами командной строки вызывается другая функция CreateProcess.

Вывод на печать

32-разрядные драйверы принтеров в 64-разрядной версии Windows использоваться не могут. Драйверы печати должны быть переведены в 64-разрядные версии. Но поскольку драйверы принтеров запускаются в адресном пространстве пользовательского режима запросившего их процесса и 64-разрядной версией Windows поддерживаются только исходные 64-разрядные драйверы принтеров, для поддержки вывода на печать из 32-разрядных процессов нужен специальный механизм. Это делается путем перенаправления всех функций вывода на печать программе Splwow64.exe, которая является RPC-сервером печати Wow64. Поскольку выполнение программы Splwow64 ведется в рамках 64-разрядного процесса, она может загружать 64-разрядные драйверы принтеров.

Ограничения

Wow64 не поддерживает выполнение 16-разрядных приложений (поддерживается на 32-разрядных версиях Windows) или загрузку 32-разрядных драйверов устройств режима ядра (они должны быть переведены в 64-разрядные версии). Wow64-процессы могут загружать только 32-разрядные DLL-библиотеки и не

[1] Подробности можно найти в разделе «Supporting 32-Bit I/O in Your 64-Bit Driver» MSDN-документации.

могут загружать исходные 64-разрядные DLL-библиотеки. Точно так же исходные 64-разрядные процессы не могут загружать 32-разрядные DLL-библиотеки. Одним из исключений является возможность загрузки ресурсов или кросс-архитектурных DLL-библиотек, состоящих из одних только данных, что допускается по той причине, что такие DLL-библиотеки содержат только данные, а не код.

В дополнение к вышесказанному, из-за разницы в размерах страниц структура Wow64 на системах IA64 не поддерживает функции ReadFileScatter, WriteFileGather, GetWriteWatch, AVX-регистры, а также технологии XSAVE и AWE. Недоступно также аппаратное ускорение через DirectX. (Для процессов Wow64 предоставляется программная эмуляция.)

Отладка в пользовательском режиме

Поддержка отладки в пользовательском режиме разбита на три разных модуля. Первый модуль расположен в самой исполняющей системе и имеет префикс Dbgk, который означает «среда отладки» — Debugging Framework. Он предоставляет все необходимые внутренние функции для регистрации и прослушивания событий отладчика, управления объектом отладки и упаковки информации с целью ее потребления его партнерами, работающими в пользовательском режиме. Компоненты пользовательского режима, общающиеся непосредственно с Dbgk, расположены в исходной системной библиотеке Ntdll.dll, в составе набора API-функций, имена которых начинаются с префикса DbgUi. Эти API-функции отвечают за создание оболочки вокруг реализации основных объектов отладки (обладающей непроницаемостью), и они позволяют всем приложениям подсистем использовать отладку, завертывая DbgUi-реализацию в свои собственные API-функции. И наконец, третий компонент в отладке пользовательского режима принадлежит DLL-библиотекам подсистем. Это открытые, документированные API-функции (из библиотеки KernelBase.dll для подсистемы Windows), которые каждая подсистема поддерживает для осуществления отладки других приложений.

Поддержка со стороны ядра

Ядро поддерживает отладку в пользовательском режиме через упомянутый ранее объект отладки. Он предоставляет серию системных вызовов, большинство из которых отображаются непосредственно на имеющиеся в Windows API-функции отладки, сначала доступные, как правило, через уровень DbgUi. Сам по себе объект отладки является довольно простой конструкцией, состоящей из серии флагов, определяющих состояние, события для уведомления любого ожидающего потока о присутствии событий отладчика, списка с двойной связью с событиями отладки, ожидающими обработки, и быстрого мьютекса, используемого для блокировки объекта. Это вся информация, которая требуется ядру для успешного получения и отправки событий отладки, и каждый подвергаемый отладке процесс имеет в своей структуре элемент порта отладки, указывающий на этот объект отладки.

После того как у процесса появится связанный с ним порт отладки, события, указанные в табл. 3.23, могут стать причиной вставки события отладки в список событий.

Таблица 3.23. События отладки режима ядра

Идентификатор события	Его значение	Чем вызывается
DbgKmExceptionApi	Произошло исключение	KiDispatchException в ходе исключения, произошедшего в пользовательском режиме
DbgKmCreate-ThreadApi	Был создан новый поток	Запуск потока пользовательского режима
DbgKmCreate-ProcessApi	Был создан новый процесс	Запуск потока пользовательского режима, являющегося первым потоком процесса
DbgKmExit-ThreadApi	Произошел выход из потока	Завершение потока пользовательского режима
DbgKmExit-ProcessApi	Произошел выход из процесса	Завершение потока пользовательского режима, который был последним потоком процесса
DbgKmLoadDllApi	Была загружена DLL-библиотека	NtMapViewOfSection, когда раздел является файлом образа (может также быть с расширением EXE)
DbgKmUnloadDllApi	DLL-библиотека была выгружена	NtUnmapViewOfSection, когда раздел является файлом образа (может также быть с расширением EXE)
DbgKmError-ReportApi	Исключение должно быть направлено в Windows Error Reporting (WER)	KiDispatchException в ходе исключения, произошедшего в пользовательском режиме, после того как отладчик не смог с ним справиться

Кроме случаев, упомянутых в таблице, есть еще пара специальных пусковых случаев за пределами обычных сценариев, которые происходят в то время, когда подвергаемый отладке объект сначала становится связанным с процессом. Первые сообщения о создании процесса *create process* и создании потока *create thread* будут отправлены не в автоматическом режиме при подключении отладчика, первое сообщение будет отправлено для самого процесса и его основного потока, а последующие — путем создания сообщения о потоках для всех других потоков в процессе. И наконец будут отправлены сообщения и событиях загрузки DLL-библиотек *load dll* для исполняемой программы, подвергаемой отладке (Ntdll.dll), а затем о всех DLL-библиотеках, загруженных в процесс, подвергаемый отладке.

После того как объект отладчика устанавливает связь с процессом, все потоки в процессе приостанавливаются. С этого момента ответственность за запуск запросов на отправку событий отладки возлагается на отладчик. Отладчики запрашивают обратную отправку событий отладки в пользовательский режим, выполняя *ожидание* на объекте отладки. Это приводит к циклическому перебору списка событий отладки. По мере того как каждый запрос удаляется из списка, его содержимое переводится из внутренней *dbgk*-структуры в структуру, *привычную* и понятную для следующего вышестоящего уровня. Как вы увидите,

эта структура также отличается и от структуры Win32, и должен быть проведен еще один уровень преобразования. Даже после обработки отладчиком всех отложенных сообщений отладки ядро не возобновляет выполнение процесса в автоматическом режиме. Чтобы возобновить выполнение, отладчик должен вызвать функцию ContinueDebugEvent.

Если не брать в расчет некоторые сложности, возникающие в силу конкретных вопросов многопоточности, базовая модель среды отладки сводится к *поставщикам* — коду в ядре, генерирующему события отладки, показанные в предыдущей таблице, и к *потребителям* — ожиданиям отладчика наступления этих событий и подтверждениям об их получении.

Встроенная поддержка

Хотя основной протокол для отладки в пользовательском режиме довольно прост, он не пригоден для непосредственного использования Windows-приложениями, вместо этого он заключается в оболочку DbgUi-функциями из библиотеки Ntdll.dll. Эта абстракция нужна для того, чтобы позволить пользоваться этими процедурами, как исходным приложениям, так и различным подсистемам (поскольку код внутри библиотеки Ntdll.dll не имеет никаких зависимостей). Функции, предоставляемые данным компонентом, во многом аналогичны Windows API-функциям и связанными с ними системным вызовам. Внутри код также предоставляет функциональные возможности, необходимые для создания объекта отладки, связанного с потоком. Дескриптор создаваемого объекта отладки никогда не открывается. Вместо этого он сохраняется в блоке окружения потока — thread environment block (TEB) — того потока отладчика, который осуществляет присоединение (см. главу 5). Это значение сохраняется в DbgSsReserved[1].

Когда отладчик присоединяется к процессу, ожидается, что процесс будет разбит на части, то есть должна произойти операция int 3 (установка контрольной точки), сгенерированная потоком, вставленным в процесс. Если этого не случится, отладчик никогда не сможет взять контроль над процессом и будет просто наблюдать за пролетающими мимо событиями отладки. За создание этого потока и его внедрение в нужный процесс отвечает библиотека Ntdll.dll.

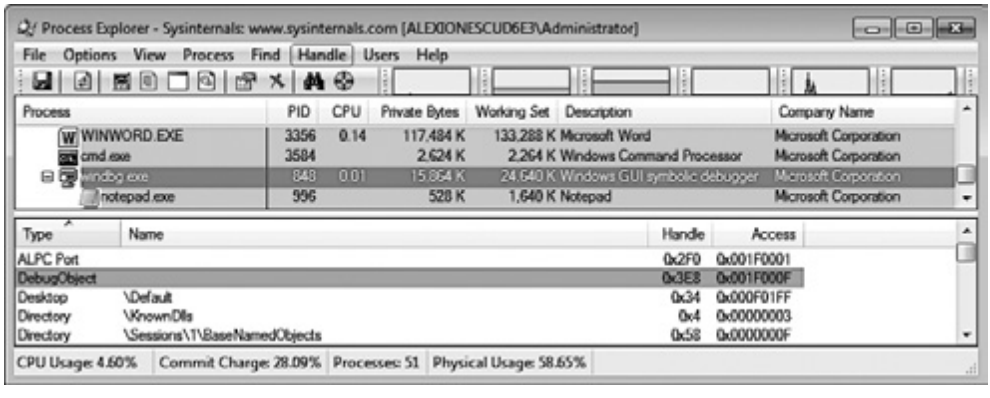
И наконец, библиотека Ntdll.dll также предоставляет API-функции для преобразования исходной структуры событий отладки в структуру, понятную Windows API-функциям.

ЭКСПЕРИМЕНТ: ПРОСМОТР ОБЪЕКТОВ ОТЛАДЧИКА

Хотя отладчик WinDbg используется для отладки в режиме ядра, его можно также использовать для отладки программ, работающих в пользовательском режиме. Начните с попытки запустить программу Блокнот — Notepad.exe с присоединенным к ней отладчиком, для чего выполните следующие действия:

1. Запустите WinDbg и щелкните на пунктах File (Файл), Open Executable (Открыть исполняемый файл).
2. Перейдите в каталог \Windows\System32\ и выберите файл Notepad.exe.
3. Отладка не входит в наши планы, поэтому все, что может появляться, нужно просто игнорировать. Чтобы заставить WinDbg продолжить выполнение программы Блокнот, можно в командном окне ввести команду g.

4. Теперь запустите Process Explorer и включите нижнюю панель, настроив ее на отображение открытых дескрипторов. (Щелкните на пунктах View (Вид), Lower Pane View (Вид нижней панели), а затем на пункте Handles (Дескрипторы).) Вам нужно также увидеть безымянные дескрипторы, поэтому щелкните на пунктах View (Вид), Show Unnamed Handles And Mappings (Показать безымянные дескрипторы и отображения).
5. Затем щелкните на процессе Windbg.exe и посмотрите на его таблицу дескрипторов. Вы должны увидеть открытый безымянный дескриптор объекта отладки. (Чтобы быстрее отыскать нужную запись, эту таблицу можно отсортировать по типу, щелнув на заголовке Type.) Вы должны увидеть нечто подобное.



Можно попробовать щелкнуть на дескрипторе правой кнопкой мыши и закрыть его. Программа Notepad должна исчезнуть, а в WinDbg должно появиться следующее сообщение:

```
ERROR: WaitForEvent failed, NTSTATUS 0xC0000354
This usually indicates that the debuggee has been
killed out from underneath the debugger.
You can use .tlist to see if the debuggee still exists.
WaitForEvent failed
```

Следует заметить, что если вы посмотрите на описание данного кода NTSTATUS, то найдете там следующий текст: «An attempt to do an operation on a debug port failed because the port is in the process of being deleted», что означает «Попытка проведения действия с портом отладки не удалась, потому что порт находится в удаленном процессе», что, собственно, и случилось из-за того, что вы закрыли дескриптор. ■

Как видите, встроенный DbgUi-интерфейс, за исключением этой абстракции, для поддержки отладки особо не утруждается. Наиболее сложные задачи он выполняет при преобразовании между встроенной структурой отладки и структурой отладки Win32. Они включают в себя несколько дополнительных изменений в структурах.

Поддержка подсистемы Windows

Последним компонентом, отвечающим за разрешение таким отладчикам, как Microsoft Visual Studio или WinDbg, вести отладку приложений пользовательского

режима, является библиотека Kernel32.dll. Она предоставляет документированные API-функции Windows. Помимо простого преобразования одного имени функции в другое, эта сторона инфраструктуры отладки отвечает еще за одну важную управленческую задачу: управление продублированными дескрипторами файла и потока.

Вспомним, что при каждой отправке события загрузки DLL-библиотеки *load DLL* дескриптор файла образа дублируется ядром и передается в структуру события, как и в случае с дескриптором процесса исполняемого файла в ходе создания события процесса. При каждом вызове ожидания *wait* библиотека Kernel32.dll проверяет, не является ли это событием, приводящем к новому дублированию из ядра дескрипторов процесса и (или) потока. Если это так, она выделяет структуру, в которой сохраняет идентификатор процесса, идентификатор потока и дескриптор потока и (или) процесса, связанного с событием. Эта структура привязана к первому индексу массива DbgSsReserved в блоке TEB, где, как уже упоминалось, хранится дескриптор объекта отладки. Точно так же библиотека Kernel32.dll проводит проверку для событий выхода. При обнаружении такого события она «помечает» дескрипторы в структуре данных.

Когда отладчик завершает использование дескрипторов и осуществляет вызов продолжения *continue*, библиотека Kernel32.dll проводит разбор этих структур, ищет любые дескрипторы, из чьих потоков произошел выход, и закрывает дескрипторы для отладчика. В противном случае выход из таких потоков и процессов никогда не произойдет, поскольку их дескрипторы будут открыты, пока работает отладчик.

Загрузчик образов

Когда на системе запускается процесс, ядро создает для его представления *объект процесса* (см. главу 5) и выполняет различные задачи инициализации, связанные с ядром. Но эти задачи не приводят к выполнению приложения, они только готовят его контекст и окружение. В действительности, в отличие от драйверов, являющихся кодом режима ядра, приложения выполняются в пользовательском режиме. Поэтому основная часть работы по инициализации проводится вне ядра. Эта работа выполняется *загрузчиком образов*, на который внутри системы ссылаются как на *Ldr*.

Загрузчик образов находится в системной DLL-библиотеке пользовательского режима Ntdll.dll и в библиотеке ядра не фигурирует. Поэтому он ведет себя как стандартный код, являющийся частью DLL-библиотеки, и на него распространяются те же ограничения относительно доступа к памяти и прав в системе безопасности. Особенность этого кода состоит в том, что он гарантированно всегда находится в запущенном процессе (библиотека Ntdll.dll всегда находится в загруженном состоянии), и в том, что первый запускаемый в пользовательском режиме фрагмент кода запускается как часть нового приложения[1].

Поскольку загрузчик запускается до самого кода приложения, он обычно остается невидимым для пользователей и разработчиков. Кроме того, при всей скрытости инициализационных задач загрузчика программа обычно при своем выполнении взаимодействует с его интерфейсом, например, при каждой загрузке

[1] Когда система создает исходный контекст, счетчик команд или указатель команд устанавливается на функцию инициализации внутри Ntdll.dll. Дополнительные сведения даны в главе 5.

или выгрузке DLL-библиотеки или при запросе базового адреса такой библиотеки. Загрузчик отвечает за выполнение следующих основных задач:

- инициализация состояния приложения в пользовательском режиме, например создание исходной кучи (динамически размещаемой структуры данных) и настройка слотов локальной памяти потока — thread-local storage (TLS) — и локальной памяти волокна (FLS);
- разбор таблицы импорта адресов — import table (IAT) — приложения для поиска всех требуемых приложением DLL-библиотек (а затем для рекурсивного разбора IAT каждой DLL), за которым следует разбор экспортной таблицы DLL-библиотек, чтобы убедиться в том, что функция уже присутствует (специальные *записи продвижения данных*, forwarder entries, могут также перенаправить экспорт на другую DLL-библиотеку);
- загрузка и выгрузка DLL-библиотек во время выполнения приложения, а также по требованию и ведение списка всех загруженных модулей (базы данных модулей);
- учет поддержки исправлений времени выполнения (так называемых горячих исправлений — hotpatching), рассматриваемых далее в этой главе;
- обработка файлов манифеста;
- чтение базы данных совместимости приложения для любых прокладок и загрузка, если требуется, DLL-библиотеки механизма прокладки;
- разрешение поддержки API-наборов и API-перенаправлений, основной части усилий MinWin по перестройке программного кода;
- разрешение динамического смягчения совместимости в процессе выполнения с использованием механизма SwitchBranch.

Как видите, большинство этих задач играет важную роль в разрешении приложению запускать его код. Без них все, начиная с вызова внешних функций и заканчивая использованием кучи, приведет к немедленному отказу. После того как процесс будет создан, загрузчик вызывает специальную встроенную API-функцию для продолжения выполнения на основе контекстного фрейма, находящегося в стеке. Этот контекстный фрейм, созданный ядром, содержит точку входа в приложение. Следовательно, поскольку загрузчик не использует стандартный вызов или безусловный переход в запущенное приложение, вы никогда не увидите инициализационную функцию загрузчика в виде части дерева вызовов в трассировке стека потока.

ЭКСПЕРИМЕНТ: НАБЛЮДЕНИЕ ЗА РАБОТОЙ ЗАГРУЗЧИКА ОБРАЗОВ

В этом эксперименте будут показаны глобальные флаги для включения свойства отладки, которое называется снимками загрузчика — *loader snaps*. Это позволит просмотреть вывод отладки из загрузчика образов при отладке запуска приложения.

1. Из каталога, в котором находится отладчик WinDbg, запустите приложение Gflags.exe, а затем щелкните на вкладке Image File (Файл образа).
2. В поле Image (Образ) наберите Notepad.exe, а затем нажмите клавишу Tab. Это даст возможность устанавливать флаги. Установите флаг Show Loader Snaps (Показывать снимки загрузчика), а затем щелкните на кнопке OK, чтобы освободить диалоговое окно.

3. Теперь выполните действия из врезки «Эксперимент: просмотр объектов отладчика» (с. 278) для запуска отладки приложения Notepad.exe.
4. Теперь вы увидите пару экранов отладочной информации, похожие на следующие:

```
0924:0248 @ 116983652 - LdrpInitializeProcess - INFO: Initializing process 0x924
0924:0248 @ 116983652 - LdrpInitializeProcess - INFO: Beginning execution of
          notepad.exe (C:\Windows\notepad.exe)
0924:0248 @ 116983652 - LdrpLoadDll - INFO: Loading DLL "kernel32.dll" from path
          "C:\Windows;C:\Windows\system32;C:\Windows\system;C:\Windows;
0924:0248 @ 116983652 - LdrpMapDll - INFO: Mapped DLL "kernel32.dll" at address
           76BD000
0924:0248 @ 116983652 - LdrGetProcedureAddressEx - INFO: Locating procedure
          "BaseThreadInitThunk" by name
0924:0248 @ 116983652 - LdrpRunInitializeRoutines - INFO: Calling init routine
          76C14592 for DLL "C:\Windows\system32\kernel32.dll"
0924:0248 @ 116983652 - LdrGetProcedureAddressEx - INFO: Locating procedure
          "BaseQueryModuleData" by name
```

5. Со временем отладчик прервет выполнение где-нибудь внутри кода загрузчика, в специальном месте, где загрузчик образов проверяет, не присоединен ли к нему отладчик, и сделает из него контрольную точку. Если для продолжения выполнения нажать клавишу G, вы увидите дополнительные сообщения, поступившие от загрузчика, и появится программа Блокнот.
6. Попробуйте поработать в Блокноте, и тогда вы увидите, как те или иные операции приводят к пробуждению загрузчика. Неплохо было бы открыть диалоговое окно сохранения или открытия файла. Тем самым будет продемонстрировано, что загрузчик запускается не только при запуске программы, но и постоянно реагирует на запросы потока, что может привести к отложенным загрузкам других модулей (которые затем могут быть выгружены после использования). ■

Ранняя стадия инициализации процесса

Поскольку загрузчик находится в библиотеке Ntdll.dll, которая является встроенной DLL-библиотекой, не связанной ни с какой конкретной подсистемой, все процессы подчинены одному и тому же поведению загрузчика (с некоторыми незначительными отличиями). В главе 5 будут подробно рассмотрены этапы, приводящие к созданию процесса в режиме ядра, а также некоторая работа, производимая Windows-функцией создания процесса CreateProcess. А здесь будет рассмотрена работа, совершаемая в пользовательском режиме, независимо от какой-либо подсистемы, как только начнет выполняться первая инструкция пользовательского режима. При запуске процесса загрузчик выполняет следующие действия:

1. Создает для приложения путевое имя образа и запрашивает для приложения раздел настроек выполнения образа файла — Image File Execution Options, а также настройки DEP- и SEH-проверок компоновщика.
2. Смотрит в заголовок исполняемого файла, чтобы определить, не является ли он .NET-приложением (на что указывает присутствие каталога образов, характерных для .NET-приложений).

3. Инициализирует для локализации процесса таблицы поддержки национальных языков — National Language Support tables (NLS).
4. Инициализирует механизм Wow64, если образ является 32-разрядным приложением и запускается на 64-разрядной версии Windows.
5. Загружает любые конфигурационные настройки, указанные в заголовке исполняемого файла. Эти настройки, которые разработчик может определить при компиляции приложения, управляют поведением исполняемого файла.
6. Устанавливает маску родственности, если таковая была указана в заголовке исполняемого файла.
7. Инициализирует FLS и TLS.
8. Инициализирует для процесса диспетчер кучи и создает первую кучу процесса.
9. Размещает для процесса контекст активации сборок SxS (Side-by-Side Assembly)/Fusion. Это позволяет системе использовать файлы той версии DLL-библиотеки, которая соответствует приложению, а не DLL-библиотеку, используемую по умолчанию, которая поставляется с операционной системой (см. главу 5).
10. Открывает каталог объектов \KnownDlls и создает путь к известным DLL. Для процесса Wow64 вместо этого каталога используется каталог \KnownDlls32.
11. Определяет текущий каталог процесса и путь для загрузки, используемый по умолчанию (используется при загрузке образов и открытии файлов).
12. Создает первые записи в таблице данных загрузчика для исполняемых приложений и Ntdll.dll и вставляет их в базу данных модулей.

Теперь загрузчик образов готов к разбору таблицы импорта исполняемых файлов, принадлежащей приложению, и приступает к загрузке любых DLL-библиотек, которые были динамически скомпонованы во время компиляции приложения. Поскольку каждая импортируемая DLL-библиотека может также иметь свою собственную таблицу импорта, эта операция будет продолжаться в рекурсивном режиме до тех пор, пока не будут удовлетворены запросы всех DLL-библиотек и не будут найдены все импортируемые функции. По мере загрузки каждой DLL-библиотеки загрузчик будет сохранять для нее информацию состояния и создавать базу данных модулей.

Разрешение имен DLL-библиотек и перенаправление

Разрешение имен является процессом, с помощью которого система преобразует имя двоичного файла PE-формата в имя физического файла в ситуациях, когда вызывающий модуль не указал или не может указать уникальный идентификатор файла. Поскольку размещение различных каталогов (каталога приложения, системного каталога и т. д.) во время компоновки не может быть жестко задано, этот процесс включает разрешение всех двоичных зависимостей, а также операций LoadLibrary, в которых вызывающий модуль не указал полный путь.

При разрешении двоичных зависимостей основная модель Windows-приложений размещает файлы в пути поиска, представляющего собой список тех мест, в которых ведется последовательный поиск файлов с соответствующим базовым именем, хотя различные системные компоненты отвергают механизм пути поиска в целях расширения модели приложения, используемой по умолчанию. Понятие

пути поиска является пережитком эпохи командной строки, когда текущий каталог приложения был важным понятием, для современных приложений с графическим интерфейсом это представляется неким анахронизмом.

Но размещение текущего каталога в таком порядке позволяет замещать операции загрузки системных двоичных файлов путем установки наносящих вред двоичных файлов с такими же базовыми именами в текущий каталог приложения. Чтобы предотвратить риски безопасности, связанные с подобным поведением, начиная с версии Windows XP SP2 появилось свойство, разрешенное по умолчанию для всех процессов и известное как безопасный режим поиска DLL-библиотек, которое было добавлено к вычислению пути поиска. Согласно безопасному режиму поиска текущий каталог перемещается за тремя системными каталогами, что приводит к следующему упорядочению пути:

1. Каталог, из которого было запущено приложение.
2. Исходный каталог системы Windows (например, C:\Windows\System32).
3. Каталог 16-разрядной системы Windows (например, C:\Windows\System).
4. Каталог Windows (например, C:\Windows).
5. Текущий каталог на момент запуска приложения.
6. Любые каталоги, указанные в переменной среды окружения %PATH%.

Для каждой последующей операции загрузки DLL-библиотеки путь поиска DLL вычисляется заново. Алгоритм, используемый для вычисления пути поиска, аналогичен алгоритму, используемому для вычисления пути поиска по умолчанию, но приложение может изменить конкретные элементы пути, отредактировав значение переменной %PATH% с помощью API-функции SetEnvironmentVariable, изменив текущий каталог с помощью API-функции SetCurrentDirectory или указав для процесса каталог DLL-библиотеки с помощью API-функции SetDllDirectory. Когда указывается каталог DLL-библиотеки, этот каталог заменяет в пути поиска текущий каталог, и загрузчик игнорирует безопасный режим поиска DLL, установленный для процесса.

Взывающие модули могут также изменить путь поиска DLL для конкретных операций загрузки, предоставив APL-функции LoadLibraryEx флаг LOAD_WITH_ALTERED_SEARCH_PATH. Если предоставлен этот флаг и в предоставленном APL-функции имени DLL указывается полная строка поиска, при вычислении пути поиска для операции вместо каталога приложения используется путь, содержащий DLL-файл.

Перенаправление имени DLL

Перед попыткой разрешения строки имени DLL в имя файла загрузчик пытается применить правила перенаправления имени DLL. Эти правила перенаправления используются для расширения или замены частей пространства имен DLL, которое обычно соответствует пространству имен файловой системы Win32 с целью расширения модели Windows-приложений. В порядке применения эти правила имеют следующий вид:

- **Перенаправление набора API-функций MinWin.** Механизм набора API-функций разработан с той целью, чтобы дать возможность команде Windows внести

изменения в исполняемый двоичный файл, экспортирующий данную системную API-функцию таким способом, который был бы прозрачен для приложений.

- **.LOCAL-перенаправление.** Механизм .LOCAL-перенаправления позволяет приложениям перенаправлять все загрузки, связанные с указанным базовым именем DLL, независимо от того, был ли указан полный путь, на локальную копию DLL в каталоге приложения. Это делается либо путем создания копии DLL с таким же базовым именем, за которым следует расширение .local (например, MyLibrary.dll.local), либо путем создания файловой папки с именем .local в каталоге приложения и помещения копии локальной DLL-библиотеки в эту папку (например, C:\Program Files\My App\.LOCAL\MyLibrary.dll). DLL-библиотеки, загрузка которых перенаправлена с помощью механизма .LOCAL, обрабатываются точно так же, как и те, для которых использовался механизм перенаправления SxS. (См. следующий пункт списка.) Загрузчик принимает .LOCAL-перенаправление загрузки DLL-библиотек только когда исполняемый файл не имеет связанного с ним либо встроенного, либо внешнего манифеста.
- **Перенаправление, связанное со слиянием (Fusion (SxS) Redirection).** Перенаправление, связанное со слиянием (также обозначаемое как совместное — side-by-side, или SxS), является расширением модели Windows-приложений, которое позволяет компонентам более подробно выразить информацию о зависимости исполняемых двоичных файлов (обычно она касается версий этих файлов) путем встраивания ресурсов, известных как манифесты. Механизм перенаправления, связанного со слиянием, был впервые использован с таким расчетом, чтобы приложения могли загрузить правильную версию пакета общих элементов управления Windows — Windows common controls package (comctl32.dll), — после того как исполняемые двоичные файлы были разбиты на разные версии, которые могут быть установлены рядом друг с другом; другие исполняемые двоичные файлы с тех пор стали получать версии аналогичным образом. Начиная с Visual Studio 2005, приложения, созданные с помощью компоновщика Microsoft linker, будут использовать Fusion для нахождения соответствующей версии C-библиотек времени выполнения.

  Инструментарий времени выполнения Fusion читает встроенную информацию о зависимостях из раздела ресурсов исполняемого двоичного файла, используя загрузчик ресурсов Windows, и упаковывает информацию о зависимостях в поисковые структуры, известные как *контексты активации* (activation contexts). Система при своей загрузке и при запуске процесса создает соответственно на уровне системы и процесса исходные контексты активации; кроме этого у каждого потока есть связанный с ним стек контекста активации, со структурой контекста активации на вершине того стека, который считается активным. Стек контекста активации, имеющийся у каждого потока, управляется как явным образом посредством API-функций ActivateActCtx и DeactivateActCtx, так и неявным образом системой в определенные моменты, например когда вызывается основная процедура DLL, являющаяся исполняемым двоичным файлом со встроенной информацией о зависимостях. Кода в рамках перенаправления осуществляется поиск имени DLL, связанного со слиянием, система ищет информацию о перенаправлении в контексте акти-

вации на вершине принадлежащего потоку стека контекста активации, а затем в контекстах активации процесса и системы; при наличии информации о перенаправлении для операции загрузки используется идентичность файла, указанная контекстом активации.

- **Известное DLL-перенаправление.** Известные DLL-библиотеки являются тем механизмом, который отображает определенные базовые имена DLL на файлы в системном каталоге, препятствуя замене DLL альтернативной версией, находящейся в другом месте.

  Одним из крайних случаев алгоритма поиска пути DLL является проверка версии DLL, осуществляемая в 64-разрядных приложениях и в приложениях WOW64. Если DLL с соответствующим базовым именем обнаружена, но впоследствии определена, как скомпилированная для неподходящей машинной архитектуры, например 64-разрядный образ в 32-разрядном приложении, загрузчик игнорирует ошибку и продолжает операцию поиска пути, начиная с элемента пути, который находится после элемента, использовавшегося для обнаружения неподходящего файла. Такой стиль поведения был разработан, чтобы дать возможность приложениям указывать в глобальной переменной среды окружения %PATH% записи как для 64-разрядных, так и для 32-разрядных вариантов.

ЭКСПЕРИМЕНТ: НАБЛЮДЕНИЕ ПОРЯДКА ПОИСКА ПРИ ЗАГРУЗКЕ DLL

Для наблюдения за тем, как загрузчик ищет DLL-библиотеки, можно воспользоваться средством Process Monitor из арсенала Sysinternals. Когда загрузчик пытается разрешить DLL-зависимость, вы увидите, что он выполняет вызов функции CreateFile, чтобы исследовать каждое место поисковой последовательности до тех пор, пока либо будет найдена указанная DLL, либо загрузка потерпит неудачу.

Рассмотрим копию экрана, отображающую поиски загрузчика, когда исполняемый файл Myapp.exe имеет статическую зависимость от библиотеки Mylibrary.dll. Исполняемый файл хранится в каталоге C:\Myapp, но при запуске исполняемого файла текущим рабочим каталогом был C:\. В демонстрационных целях исполняемый файл не включает в себя какой-либо манифест (по умолчанию у Visual Studio есть один манифест), поэтому загрузчик будет вести проверку внутри подкаталога C:\Myapp\Myapp.exe.local, который был создан для проведения эксперимента. Чтобы уменьшить объем помех, фильтр Process Monitor включает процесс myapp.exe и все части, в которых содержится строка «mylibrary.dll».

Process Name	Operation	Path	Result
myapp.exe	CreateFile	C:\myapp\myapp.exe.local\mylibrary.dll	NAME NOT FOUND
myapp.exe	CreateFile	C:\myapp\mylibrary.dll	NAME NOT FOUND
myapp.exe	CreateFile	C:\myapp\mylibrary.dll	NAME NOT FOUND
myapp.exe	CreateFile	C:\Windows\SysWOW64\mylibrary.dll	NAME NOT FOUND
myapp.exe	CreateFile	C:\Windows\system\mylibrary.dll	NAME NOT FOUND
myapp.exe	CreateFile	C:\Windows\mylibrary.dll	NAME NOT FOUND
myapp.exe	CreateFile	C:\mylibrary.dll	SUCCESS
myapp.exe	QueryBasicInf...	C:\mylibrary.dll	SUCCESS
myapp.exe	CloseFile	C:\mylibrary.dll	SUCCESS
myapp.exe	CreateFile	C:\mylibrary.dll	SUCCESS
myapp.exe	CreateFileMa...	C:\mylibrary.dll	FILE LOCKED WITH ONL...
myapp.exe	QueryStandar...	C:\mylibrary.dll	SUCCESS
myapp.exe	CreateFileMa...	C:\mylibrary.dll	SUCCESS
myapp.exe	Load Image	C:\mylibrary.dll	SUCCESS
myapp.exe	CloseFile	C:\mylibrary.dll	SUCCESS

Обратите внимание на соответствие порядка поиска его описанию. Сначала загрузчик, в то время когда был запущен исполняемый файл, проверяет подкаталог .LOCAL, затем каталог, где находится исполняемый файл, затем каталог C:\Windows\System32 (поскольку это 32-разрядный исполняемый файл, который перенаправляется в каталог C:\Windows\SysWOW64), затем каталог 16-разрядной версии Windows, затем C:\Windows и, наконец, текущий каталог. Событие Load Image подтверждает, что загрузчик успешно справился с импортом. ■

База данных загруженных модулей

Загрузчик ведет список всех модулей (DLL-библиотек, а также основных исполняемых файлов), которые были загружены процессом. Эта информация хранится в создаваемой для каждого процесса структуре, которая называется блоком окружения процесса — process environment block, или PEB (см. главу 5), — а именно в подструктуре, имеющей идентификатор Ldr и называемой PEB_LDR_DATA. В этой структуре загрузчик ведет три списка с двойной связью, в которых содержится одна и та же, но по-разному выстроенная информация: по порядку загрузки, по размещению в памяти или по порядку инициализации. Эти списки содержат структуры, называемые записями таблицы данных загрузчика (LDR_DATA_TABLE_ENTRY), с которых хранится информация о каждом модуле. В табл. 3.24 перечислены различные фрагменты информации, вносимой загрузчиком в запись.

Таблица 3.24. Поля в записи таблицы данных загрузчика

Поле	Значение
BaseDllName	Имя самого модуля без полного пути к нему
ContextInformation	Используется функцией SwitchBranch (рассматриваемой далее) для хранения текущего контекста GUID Windows, связанного с этим модулем
DllBase	Хранит базовый адрес, по которому был загружен модуль
EntryPoint	Содержит начальную процедуру модуля (например DllMain)
EntryPointActivation-Context	При вызове инициализаторов содержит контекст активации SxS/Fusion
Flags	Флаги состояния загрузчика для этого модуля. (Описание флагов дано в табл. 3.25.)
ForwarderLinks	Связанный список модулей, которые были загружены из модуля в результате использования механизма продвижения данных экспортной таблицы
FullDllName	Полностью указанное путевое имя модуля
HashLinks	Связанный список, используемый во время запуска и остановки процесса для более быстрого поиска
List Entry Links	Связывает данную запись с каждым из трех упорядоченных списков, являющихся частью базы данных загрузчика
LoadCount	Счетчик ссылок на модуль (показывающий, сколько раз он был загружен)

продолжение ⇩

Поле	Значение
LoadTime	Хранит показания системного времени на момент загрузки этого модуля
OriginalBase	Хранит исходный базовый адрес модуля (установленный компоновщиком), позволяющий быстрее обрабатывать перемещенные записи импорта
PatchInformation	Информация, имеющая важное значение в ходе операции внесения горячих исправлений (hotpatch) в этот модуль
ServiceTagLinks	Связанный список служб (дополнительные сведения даны в главе 4), ссылающихся на этот модуль
SizeOfImage	Размер модуля в памяти
StaticLinks	Связанный список модулей, загруженных в результате статических ссылок из данного модуля
TimeDateStamp	Отметка времени, записанная компоновщиком при сборке модуля, которую загрузчик получает из PE-заголовка образа модуля
TlsIndex	Слот локального хранилища потока, связанного с данным модулем

Одним из способов просмотра базы данных загрузчика процесса является использование средства WinDbg и его отформатированного вывода блока окружения процесса — PEB. В следующем эксперименте показано, как это сделать и как самому посмотреть на структуру LDR_DATA_TABLE_ENTRY.

ЭКСПЕРИМЕНТ: ВЫВОД ДАМПА БАЗЫ ДАННЫХ ЗАГРУЖЕННЫХ МОДУЛЕЙ

Перед началом этого эксперимента выполните те же действия, которые производились в двух предыдущих экспериментах по запуску программы Notepad.exe с использованием в качестве отладчика WinDbg. Когда дойдете до первого приглашения на ввод данных (там, где до этого момента вам предписывалось ввести команду g), выполните следующие инструкции:

1. PEB-блок текущего процесса можно просмотреть с помощью команды !peb. Но теперь нас интересуют только выводимые на экран данные Ldr (см. главу 5).

```
0: kd> !peb
PEB at 000007fffffda000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            No
    ImageBaseAddress:         00000000ff590000
    Ldr                       0000000076e72640
    Ldr.Initialized:          Yes
    Ldr.InInitializationOrderModuleList: 0000000000212880 . 0000000004731c20
    Ldr.InLoadOrderModuleList:           0000000000212770 . 0000000004731c00
    Ldr.InMemoryOrderModuleList:         0000000000212780 . 0000000004731c10
            Base TimeStamp                     Module
        ff590000 4ce7a144 Nov 20 11:21:56 2010 C:\Windows\Explorer.EXE
```

```
    76d40000 4ce7c8f9 Nov 20 14:11:21 2010 C:\Windows\SYSTEM32\ntdll.dll
    76870000 4ce7c78b Nov 20 14:05:15 2010 C:\Windows\system32\kernel32.dll
7fefd2d0000 4ce7c78c Nov 20 14:05:16 2010 C:\Windows\system32\KERNELBASE.dll
7fefee20000 4a5bde6b Jul 14 02:24:59 2009 C:\Windows\system32\ADVAPI32.dll
```

2. Адрес, показанный в строке Ldr, является указателем на ранее рассмотренную структуру PEB_LDR_DATA. Обратите внимание на то, что WinDbg показывает вам адрес трех списков и выводит дамп списка модулей в порядке их инициализации, где показан полный путь, отметка времени и базовый адрес каждого модуля.
3. Можно также проанализировать отдельно запись каждого модуля путем прохода по списку модулей с последующим выводом дампа данных по каждому адресу, отформатированных в виде структуры LDR_DATA_TABLE_ENTRY. Но, вместо того чтобы делать это для каждой записи, WinDbg может проделать основную часть работы, используя расширение !list и следующий синтаксис:

```
!list –t ntdll!_LIST_ENTRY.Flink –x "dt ntdll!_LDR_DATA_TABLE_ENTRY @$extret\"
0000000076e72640
```

Обратите внимание на то, что последнее число не является постоянным: оно зависит от того, что показано на вашей машине в Ldr.InLoadOrderModuleList.

4. Затем вы можете посмотреть записи для каждого модуля:

```
0:001> !list -t ntdll!_LIST_ENTRY.Flink -x "dt ntdll!_LDR_DATA_TABLE_ENTRY
@$extret\" 001c1cf8
   +0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x1c1d68 - 0x76fd4ccc ]
   +0x008 InMemoryOrderLinks : _LIST_ENTRY [ 0x1c1d70 - 0x76fd4cd4 ]
   +0x010 InInitializationOrderLinks : _LIST_ENTRY [ 0x0 - 0x0 ]
   +0x018 DllBase          : 0x00d80000
   +0x01c EntryPoint       : 0x00d831ed
   +0x020 SizeOfImage      : 0x28000
   +0x024 FullDllName      : _UNICODE_STRING "C:\Windows\notepad.exe"
   +0x02c BaseDllName      : _UNICODE_STRING "notepad.exe"
   +0x034 Flags            : 0x4010
```

Хотя в этом разделе рассматривается загрузчик пользовательского режима, который находится в библиотеке Ntdll.dll, обратите внимание на то, что ядро также использует свой собственный загрузчик для драйверов и зависимых DLL-библиотек, с похожей структурой записей. Точно так же у загрузчика режима ядра есть своя собственная база данных каждой записи, которая напрямую доступна через глобальную переменную PsActiveModuleList. Для вывода дампа базы данных модулей, загруженных в ядро, можно воспользоваться такой же командой !list, которая была показана в предыдущем эксперименте, заменив указатель в конце команды строкой «nt!PsActiveModuleList».

Просмотр списка этого необработанного формата позволяет еще глубже разобраться во внутренностях загрузчика, таких как поля флагов, содержащие информацию о состоянии, которую сама по себе команда !peb показать не в состоянии. Значения флагов показаны в табл. 3.25. Поскольку эту структуру используют оба загрузчика, и ядра и пользовательского режима,

некоторые флаги применимы только к драйверам режима ядра, в то время как другие флаги применимы только к приложениям пользовательского режима (например, связанные с .NET-состоянием).

Таблица 3.25. Флаги записей таблицы данных загрузчика

Флаг	Значение
LDRP_STATIC_LINK (0x2)	Этот модуль востребован, потому что на него есть ссылка в таблице импорта
LDRP_IMAGE_DLL (0x4)	Модуль является DLL образа (а не DLL данных или исполняемым файлом)
LDRP_IMAGE_INTEGRITY_FORCED (0x20)	Модуль был связан с /FORCEINTEGRITY (содержит в своем PE-заголовке IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY_)
LDRP_LOAD_IN_PROGRESS (0x1000)	Этот модуль загружается в данный момент
LDRP_UNLOAD_IN_PROGRESS (0x2000)	Этот модуль выгружается в данный момент
LDRP_ENTRY_PROCESSED (0x4000)	Загрузчик завершил обработку данного модуля
LDRP_ENTRY_INSERTED (0x8000)	Загрузчик завершил вставку этой записи в базу данных загруженных модулей
LDRP_FAILED_BUILTIN_LOAD (0x20000)	Показывает, что при загрузке системы этот драйвер загрузить не удалось
LDRP_DONT_CALL_FOR_THRAS (0x40000)	Уведомления DLL_THREAD_ATTACH/DETACH этой DLL отправлять не нужно
LDRP_PROCESS_ATTACH_CALLED (0x80000)	Эта DLL отправила уведомление DLL_PROCESS_ATTACH
LDRP_DEBUG_SYMBOLS_LOADED (0x100000)	Символы отладки для этого модуля были загружены ядром или пользовательским отладчиком
LDRP_IMAGE_NOT_AT_BASE (0x200000)	Этот образ был перемещен из его исходного базового адреса
LDRP_COR_IMAGE (0x400000)	Этот модуль является .NET-приложением
LDRP_COR_OWNS_UNMAP (0x800000)	Отображение этого модуля должно быть удалено .NET-библиотекой времени выполнения
LDRP_SYSTEM_MAPPED (0x1000000)	Этот модуль отображен на адресное пространство ядра с помощью записей таблицы страниц системы — System PTE (а не находится в памяти исходного пускового загрузчика)
LRP_IMAGE_VERIFYING (0x2000000)	Этот модуль в данный момент проверяется с помощью средства Driver Verifier
LDRP_DRIVER_DEPENDENT_DLL (0x4000000)	Этот модуль является DLL, которая находится в таблице импорта, принадлежащей драйверу
LDRP_ENTRY_NATIVE (x000000)	Этот модуль был скомпилирован для Windows 2000 или более поздних версий. Этот флаг используется средством Driver Verifier в качестве свидетельства о том, что драйвер может попасть под подозрение

Флаг	Значение
LDRP_REDIRECTED (0x10000000)	В файле манифеста указан файл перенаправления для этой DLL
LDRP_NON_PAGED_DEBUG_INFO (0x20000000)	Отладочная информация для этого модуля находится в невыгружаемой памяти
LDRP_MM_LOADED (0x40000000)	Этот модуль был загружен загрузчиком ядра с помощью функции MmLoadSystemImage
LDRP_COMPAT_DATABASE_PROCESSED (0x80000000)	Эта DLL была обработана механизмом прокладки

Анализ импорта

После того как был рассмотрен способ отслеживания загрузчиком всех модулей, загруженных для процесса, можно продолжить анализ инициализационных задач, выполняемых загрузчиком при запуске. На этой стадии загрузчик выполняет следующие действия:

1. Загружает каждую DLL, на которую есть ссылка в таблице импорта исполняемого процессом образа.
2. Проверяет, не была ли DLL уже загружена, просматривая для этого базу данных модуля. Если библиотека не будет найдена в списке, загрузчик открывает DLL и отображает ее в памяти.
3. В ходе операции отображения загрузчик сначала просматривает различные пути, где он должен попытаться найти эту DLL, а также выясняет, не является ли эта библиотека «известной DLL», это будет означать, что система уже загрузила ее во время запуска и предоставила для доступа к ней глобальный файл, отображенный в памяти. Могут также случиться определенные отклонения от стандартного алгоритма поиска, связанные либо с использованием файла с расширением .local (что заставит загрузчик использовать DLL-библиотеки в локальном пути), либо с файлом манифеста, который может указать на необходимость использования DLL, находящейся в другом месте, чтобы гарантировать использование конкретной версии.
4. После того как DLL найдена на диске и отображена, загрузчик проверяет, не загрузило ли ядро ее в какое-нибудь другое место — это называется перемещением. Если загрузчик обнаружил перемещение, он проводит анализ информации о перемещении в DLL и выполняет требуемые операции. Если информация о перемещении отсутствует, происходит сбой загрузки DLL.
5. Затем загрузчик создает запись таблицы данных загрузчика для этой DLL и вставляет ее в базу данных.
6. После того как DLL была отображена, процесс повторяется для этой DLL с целью анализа ее таблицы импорта и всех ее зависимостей.
7. После загрузки каждой DLL загрузчик проводит анализ IAT с целью поиска конкретных импортированных функций. Обычно это делается по именам, но также может делаться и по порядку (по порядковому номеру). Для каждого имени загрузчик проводит анализ экспортной таблицы импортированной DLL и пытается найти соответствие. Если соответствие найдено не будет, операция прерывается.

8. Таблица импорта, принадлежащая образу, может также быть связанной. Это означает, что в процессе компоновки разработчики уже назначили статические адреса, указывающие на импортируемые функции во внешних DLL-библиотеках. Это исключает потребность в поиске каждого имени, но предполагает, что те DLL-библиотеки, которые будут использоваться приложением, будут всегда находиться по одним и тем же адресам. Поскольку в Windows используется произвольное использование адресного пространства, к системным приложениям и библиотекам это, как правило, не относится.
9. Экспортная таблица импортированной DLL может использовать запись продвижения данных (forwarder entry), означающую, что текущая функция реализована в другой DLL. По сути это должно рассматриваться как импорт или зависимость, поэтому после анализа экспортной таблицы загружается также и каждая DLL, на которую ссылалась запись продвижения данных, и загрузчик возвращается к пункту 1.

После того как были загружены все импортируемые DLL-библиотеки (и их собственные зависимости или импортируемые данные), найдены все требуемые импортируемые функции и были загружены и обработаны все записи продвижения данных, этап завершается: теперь все зависимости, которые были определены приложением и его различными DLL-библиотеками во время компиляции, удовлетворены. В ходе выполнения к загрузчику могут обращаться и, по сути, повторно выполнять те же самые задачи отложенные зависимости (называемые отложенной загрузкой), а также операции времени выполнения (например, вызовы функции LoadLibrary). Но следует заметить, что отказы на данных этапах, если они выполняются во время запуска процесса, приведут к ошибке запуска приложения. Например, попытка запуска приложения, требующего функцию, отсутствующую в текущей версии операционной системы, может привести к выводу сообщения, подобного показанному на рис. 3.32.

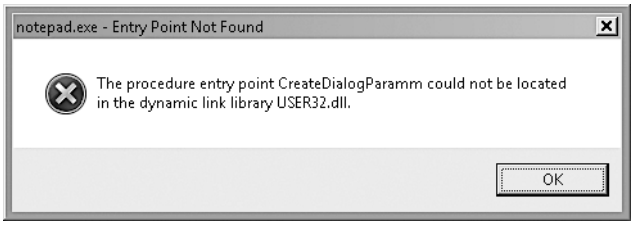


Рис. 3.32. Диалоговое окно, показываемое, когда требуемая импортируемая функция отсутствует в DLL

Инициализация процесса после импортирования

После загрузки зависимостей для полного завершения запуска приложения должен быть выполнен ряд инициализационных задач. На этой стадии загрузчик должен сделать следующее:

1. Проверить, не относится ли приложение к категории .NET-приложений, и перенаправить выполнение на точку входа .NET-системы времени выполнения, предполагая, что образ уже был проверен средой.

2. Проверить, не требует ли перемещения само приложение, и обработать для приложения записи перемещения. Если приложение не может быть перемещено или не содержит информации о перемещении, загрузка потерпит неудачу.
3. Проверить, не использует ли приложение TLS, и провести поиск в исполняемом файле приложения TLS-записей, необходимых ему для размещения и настройки.
4. Если это Windows-приложение, то после kernel32.dll размещается переходной код инициализации потока подсистемы Windows и включается обязательное выполнение Authz/AppLocker (см. главу 6). Если Kernel32.dll не найдена, выстраивается предположение, что система запущена в MinWin и загружена только библиотека Kernelbase.dll.
5. Загрузить, наконец, любой статический импорт.
6. К этому моменту при использовании такого отладчика, как WinDbg, будет достигнута исходная контрольная тока отладчика. Именно здесь для продолжения выполнения в ранее рассмотренных экспериментах вам приходилось вводить команду g.
7. Убедиться в том, что приложение будет нормально запущено, если система относится к мультипроцессорным системам.
8. Установить настройки по умолчанию предотвращения выполнения данных — data execution prevention (DEP), включая настройки для проверки цепи исключения, которая называется также «программной» DEP.
9. Проверить, не требует ли это приложение какой-либо работы по совместимости приложений, и загружает, если требуется, механизм прокладки.
10. Обнаружить возможное наличие защиты приложения с помощью SecuROM, SafeDisc и других разновидностей оболочек или защитных утилит, у которых могут быть проблемы с DEP (и перенастроить в таких случаях установки DEP).
11. Запустить инициализаторы для всех загруженных модулей.
12. Запустить функцию обратного вызова механизма прокладки (Shim Engine), предназначенную для работы по завершении инициализации, если к модулю была применена прокладка для совместимости приложений.
13. Запустить зарегистрированную в PEB процедуру инициализации, связанной с подсистемой DLL-библиотеки, запускаемую по завершении процесса. Для Windows-приложений делаются, к примеру, проверки, характерные для Службы терминалов.

Запуск инициализаторов на последнем основном этапе является работой загрузчика. На этом этапе для каждой DLL вызывается процедура DllMain, позволяя каждой DLL провести ее собственную инициализационную работу, которая даже может включать в процессе выполнения загрузку новых DLL. Кроме того, проводится обработки инициализаторов TLS каждой DLL. Это один из последних этапов, на котором может произойти сбой в загрузке приложения. Если код успешного завершения своих процедур DllMain вернули не все загружаемые DLL, загрузчик выполняет аварийное прекращение запуска приложения. На самом последнем этапе загрузчик вызывает TLS-инициализатор текущего приложения.

Технология SwitchBack

По мере внесения исправлений (например, связанных с конкуренцией и проверкой на наличие неверных параметров в существующих API-функциях) в каждую новую версию Windows для каждого изменения, даже самого незначительного, возникает риск возникновения несовместимости приложений. В Windows используется технология SwitchBack, реализованная в загрузчике, которая позволяет разработчикам программного обеспечения встроить GUID-идентификатор, зависящий от версии Windows, на которую они нацеливались в своем манифесте, связанном с исполняемым файлом. Например, если разработчик хочет воспользоваться усовершенствованиями, добавленными в Windows 7 в данные API-функции, он должен включить в свой манифест GUID-идентификатор Windows 7, а если у разработчика имеется устаревшее приложение, сориентированное на поведение, характерное для Windows Vista, он должен поместить в манифест GUID-идентификатор Windows Vista. SwitchBack проводит анализ этой информации и соотносит ее со встроенной информацией в SwitchBack-совместимых DLL-библиотеках (в разделе образа .sb_data), чтобы решить, какая из версий затронутых API-функций должна быть вызвана модулем. Поскольку технология SwitchBack работает на уровне загруженных модулей, она позволяет процессу иметь параллельные вызовы одних тех же API-функций как из устаревших, так и из текущих DLL-библиотек, следуя при этом различным результатам.

В настоящее время в Windows определены два GUID-идентификатора, представляющие установки совместимости либо с Windows Vista, либо с Windows 7:

- {e2011457-1546-43c5-a5fe-008deee3d3f0} для Windows Vista;
- {35138b9a-5d96-4fbd-8e2d-a2440225f93a} для Windows 7.

Эти GUID-идентификаторы должны присутствовать в файле манифеста приложения ниже идентификатора SupportedOS ID, присутствующего в записи атрибута совместимости. Если в манифесте приложения не содержится GUID, в качестве режима совместимости по умолчанию будет выбрана Windows Vista. Запуск в контексте Windows 7 влияет на следующие компоненты:

- RPC-компоненты вместо закрытой реализации используют пул потоков Windows.
- Блокировка DirectDraw Lock не может быть получена на первичном буфере.
- Копирование битового массива (blitting) на рабочем столе не допускается без отсечения (clipping) окна.
- Исправляется условие конкуренции в GetOverlappedResult.

Когда Windows API подвергается изменениям, которые могут нарушить совместимость, код входа в функции вызывает функцию SbSwitchProcedure, чтобы запустить SwitchBack-логику. Она проходит по указателю в таблицу модулей SwitchBack, в которой содержится информация о механизмах SwitchBack, задействованных в модуле. В таблице также содержится указатель на массив записей для каждой точки SwitchBack. В этой таблице содержится дескриптор каждой точки ветвления, который идентифицирует ее с символическим именем и с полным описанием, а также со связанным тегом смягчения совместимости. Обычно в модуле имеются две точки ветвления, одна для поведения в Windows Vista и одна для поведения в Windows 7. Для каждой точки ветвления дается требуе-

мый контекст SwitchBack, тот самый контекст, который определяет, какая из двух (или более) ветвей будет взята во время выполнения приложения. И наконец, в каждом из этих дескрипторов содержится указатель на функцию с текущим кодом, который должен выполниться при каждом ветвлении. Если приложение запускается с Windows 7 GUID, этот код будет частью его SwitchBack-контекста и API-функции SbSelectProcedure и после анализа таблицы модулей выполнит соответствующую операцию. Он находит в дескрипторе модуля запись контекста и приступает к вызову указателя функции, включенного в дескриптор.

SwitchBack использует ETW, чтобы проследить выбор данных SwitchBack-контекстов и точек ветвления, и снабжает данными регистратор Windows AIT (Application Impact Telemetry — телеметрия влияния на приложение). Эти данные могут периодически собираться компанией Microsoft для определения степени использования каждой записи совместимости, идентификации того приложения, которое ее использует (в журнале предоставляется полная трассировка стека), и для уведомления сторонних поставщиков.

Как уже упоминалось, уровень совместимости приложения хранится в его манифесте. В время загрузки загрузчик анализирует файл манифеста, создает структуру контекстных данных и помещает ее в кэш-память в качестве представителя pContextData блока окружения процесса. (Более подробно блок окружения процесса рассмотрен в главе 5.) В этих данных контекста содержатся соответствующие GUID-идентификаторы совместимости, под которыми выполняется данный процесс, и определяется, какая версия точек ветвления у вызванных API-функций, которые будут выполняться используемой технологией SwitchBack.

Наборы API-функций

Несмотря на то что SwitchBack для определенных сценариев совместимости приложений использует API-перенаправление, для всех приложений в Windows используется гораздо более распространенный механизм перенаправления, который называется наборами API-функций. Его целью является предоставление тонкого категорирования API-функций Windows в подчиненных DLL-библиотеках вместо содержания больших многоцелевых DLL-библиотек, охватывающих почти тысячи API-функций, которые могут не понадобиться всем типам ныне существующих и будущих Windows-систем. Эта технология, разработанная в основном для поддержки перестройки самых нижних уровней архитектуры Windows для отделения ее от более высоких уровней, идет рука об руку с разбиением Kernel32.dll и Advapi32.dll (наряду со всем другим) на несколько виртуальных DLL-файлов.

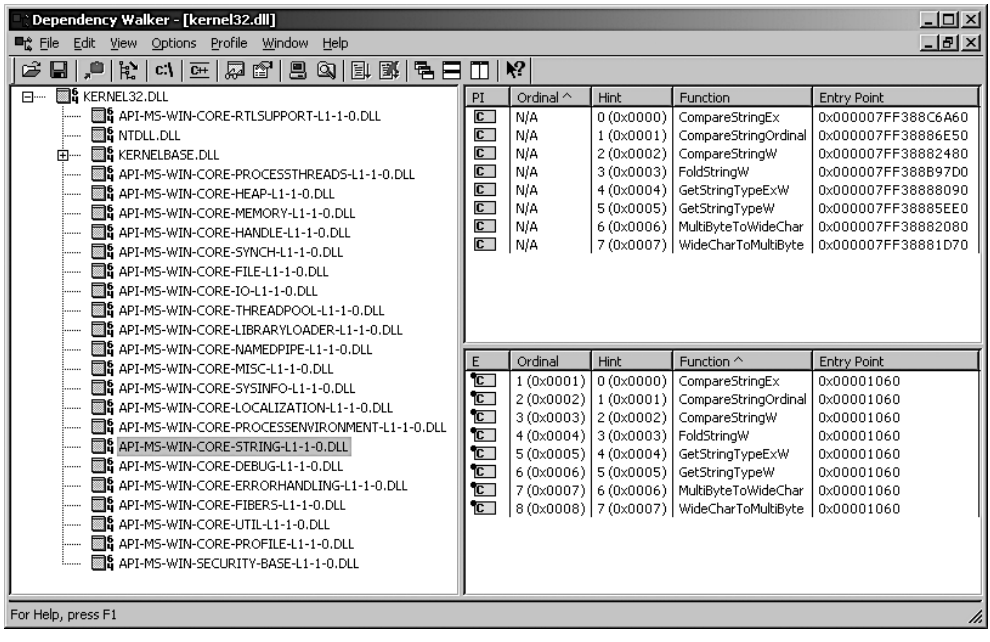
Например, на следующем изображении показывается, что Kernel32.dll, являющаяся основной библиотекой Windows, ведет импорт из множества других DLL-библиотек, начиная с API-MS-WIN. В каждой из этих DLL-библиотек содержится небольшой поднабор API-функций, которые обычно предоставляются библиотекой Kernel32, но все вместе они составляют все пространство API, открываемое библиотекой Kernel32.dll. Например, библиотека CORE-STRING содержит только основные строковые функции Windows.

За счет разбиения функций на отдельные файлы достигаются две цели:

- во-первых это позволяет будущим приложениям компоноваться только с теми API-библиотеками, которые предоставляют нужные им функции;

- во-вторых, если Microsoft создаст версию Windows, не поддерживающую, к примеру, локализацию (скажем, для встроенной системы, не имеющей выхода на пользователя и предназначенной только для англоязычных стран), она сможет просто удалить соответствующие подчиненные DLL-библиотеки и изменить схему набора API-функций. В результате будет получен более компактный двоичный код Kernel32, и любое приложение, не требующее локализации, будет работать по-прежнему.

Следуя этой технологии, была определена (реализована на уровне исходного кода) «базовая» Windows-система под названием «MinWin» с минимальным набором служб, куда включено ядро, основные драйверы (включая файловую систему, основные системные процессы, такие как CSRSS и Диспетчер управления службами (Service Control Manager)) и еще небольшой перечень Windows-служб. Семейство встраиваемых операционных систем Windows Embedded имеет специальный инструмент для генерации индивидуального ядра операционной системы, который называется Platform Builder. Это семейство предоставляет то, что может показаться аналогичной технологией, поскольку построители системы могут удалять выбранные компоненты «Windows components», такие как оболочка или сетевой стек. Но удаление компонентов из Windows оставляет зависшие зависимости, пути выполнения кода, которые, при выполнении этого кода, приведут к сбою, поскольку они зависят от удаленных компонентов. С другой стороны, зависимости MinWin являются целиком и полностью самодостаточными.



При инициализации диспетчера процессов вызывается функция PspInitializeApiSetMap, отвечающая за создание объекта раздела (используя стандартный объект раздела) таблицы перенаправления API-набора, которая хранится в %SystemRoot%\System32\ApiSetSchema.dll. Эта библиотека не содержит

исполняемого кода, но в ней есть раздел .apiset, в котором содержатся данные отображения API-набора, отображающие виртуальные DLL-библиотеки API-набора на логические DLL-библиотеки, где реализованы API-функции. При запуске нового процесса диспетчер процессов отображает объект раздела на адресное пространство процесса и устанавливает значение поля ApiSetMap в PEB-блоке процесса, чтобы указать на базовый адрес, куда был отображен объект раздела.

В свою очередь, функция загрузчика LdrpApplyFileNameRedirection, которая обычно отвечает за перенаправление рассмотренного ранее .local и SxS/Fusion манифеста, также проверяет перенаправление данных API-набора, когда загружается новая импортируемая библиотека, имя которой начинается с «API-» (как в динамическом, так и в статическом режиме). Таблица API-набора выстраивается библиотекой таким образом, что в каждой записи есть описание, в какой логической DLL-библиотеке может быть найдена та или иная функция и какая DLL-библиотека в связи с этим была загружена. Хотя у данных схемы двоичный формат, дамп строк этих данных можно вывести с помощью средства Sysinternals Strings, чтобы посмотреть, какие DLL-библиотеки определены на данный момент:

```
C:\Windows\System32>strings apisetschema.dll
...
MS-Win-Core-Console-L1-1-0
kernel32.dllMS-Win-Core-DateTime-L1-1-0
MS-Win-Core-Debug-L1-1-0
kernelbase.dllMS-Win-Core-DelayLoad-L1-1-0
MS-Win-Core-ErrorHandling-L1-1-0
MS-Win-Core-Fibers-L1-1-0
MS-Win-Core-File-L1-1-0
MS-Win-Core-Handle-L1-1-0
MS-Win-Core-Heap-L1-1-0
MS-Win-Core-Interlocked-L1-1-0
MS-Win-Core-IO-L1-1-0
MS-Win-Core-LibraryLoader-L1-1-0
MS-Win-Core-Localization-L1-1-0
MS-Win-Core-LocalRegistry-L1-1-0
MS-Win-Core-Memory-L1-1-0
MS-Win-Core-Misc-L1-1-0
MS-Win-Core-NamedPipe-L1-1-0
MS-Win-Core-ProcessEnvironment-L1-1-0
MS-Win-Core-ProcessThreads-L1-1-0
MS-Win-Core-Profile-L1-1-0
MS-Win-Core-RtlSupport-L1-1-0
ntdll.dll
MS-Win-Core-String-L1-1-0
```

Гипервизор (Hyper-V)

Одна из ключевых технологий в индустрии программного обеспечения, используемая системными администраторами, разработчиками, а также тестировщиками,

называется виртуализацией, и она относится к возможности одновременного запуска на одной и той же физической машине нескольких операционных систем. Одна операционная система, в которой выполняется программное обеспечение виртуализации, называется хостом, а другие операционные системы запускаются внутри программы виртуализации в качестве гостевых. Сценарии использования этой модели охватывают все: от возможности тестирования приложения на разных платформах и до наличия полностью виртуальных серверов, каждый из которых запускается как часть одной и той же машины и управляются из единого центра.

До недавнего времени вся виртуализация осуществлялась самим программным обеспечением, иногда с помощью технологии виртуализации аппаратного уровня (так называемой виртуализацией на основе хоста). Благодаря аппаратной виртуализации центральный процессор может осуществлять основную часть уведомлений, необходимых для перехвата инструкций и виртуализации доступа к памяти. Эти уведомления, а также различные настроечные действия, которые требуются, чтобы разрешить гостевым операционным системам выполняться параллельно, должны быть обработаны частью инфраструктуры, совместимой с поддержкой виртуализации со стороны центрального процессора. Вместо того чтобы для выполнения этих задач полагаться на отдельные части программного обеспечения, запущенного внутри операционной системы хоста, можно использовать гипервизор — ограниченную часть системного программного обеспечения низкого уровня, в полной мере использующую поддержку виртуализации со стороны аппаратного обеспечения. На рис.3.33 показан простой архитектурный обзор этих двух разновидностей систем.

Рис. 3.33. Две архитектуры для виртуализации

Используя Hyper-V, на серверные компьютеры под управлением Windows можно установить поддержку для виртуализации на основе гипервизора в роли сервера (если лицензирована версия с поддержкой Hyper-V). Поскольку гипервизор является частью операционной системы, управляющей гостевыми операционными системами внутри себя, а также взаимодействующей с ними, он полностью интегрирован в операционную систему через стандартные механизмы управления, такие как WMI и службы (см. главу 4).

И наконец, помимо гипервизора, позволяющего запускать другие гостевые операционные системы, управляемые с помощью Windows Server host, оба варианта версий Windows, и клиентский и серверный, поставляются также с *просвещениями* (enlightenments). Просвещения — средства, являющиеся специаль-

ными средствами оптимизации ядра и, возможно, драйверов устройств, которые определяют, что код был запущен в качестве гостевой операционной системы под управлением гипервизора, и выполнять определенные задачи по-разному *или более эффективно, с учетом этой среды. Часть из этих усовершенствований будут рассмотрены чуть позже, а сейчас посмотрим на базовую архитектуру стека виртуализации Windows, показанную на рис. 3.34.

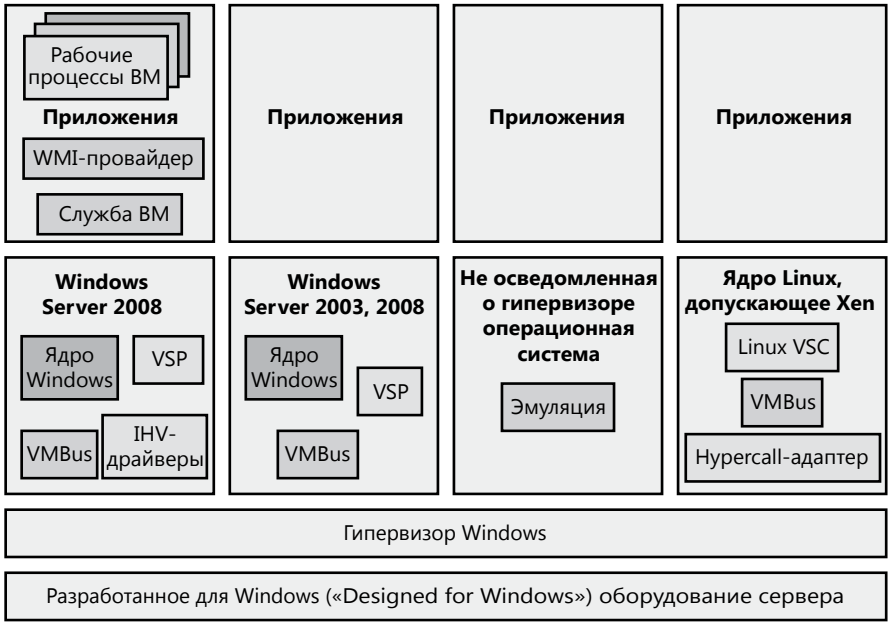


Рис. 3.34. Архитектурный стек Windows Hyper-V

Разделы

Одним из ключевых архитектурных компонентов, положенных в основу гипервизора Windows, является понятие разделов. По сути, раздел имеет отношение к экземпляру установки операционной системы, которая может быть либо тем, что традиционно называется хостом, либо относиться к гостевой операционной системе. В модели гипервизора Windows эти два понятия не употребляются, вместо них мы будем говорить, соответственно, либо о родительском, либо о дочернем разделе. Следовательно, как минимум, система Hyper-V будет иметь дочерний раздел, для которого рекомендуется иметь установку ядра Windows Server, а также стека виртуализации и связанных с ним компонентов. Хотя этот тип установки рекомендуется по той причине, что он позволяет минимизировать установку исправлений и сократить область применения системы безопасности, в результате чего повышается доступность сервера, поддерживается также и полная установка. Каждая операционная система, запущенная внутри виртуализированной среды, представляет собой дочерний раздел, который может содержать определенные дополнительные средства, оптимизирующие доступ к оборудованию или позволяющие осуществлять управление операционной системой.

Родительский раздел

Одной из основных целей при разработке гипервизора Windows было сохранение его минимальных размеров и придание ему максимально возможной модульности, создав что-то вроде микроядра, а не предоставление полнофункционального монолитного модуля. Это означает, что основная часть виртуализационной работы фактически проводится отдельным стеком виртуализации, и к тому же без каких-либо драйверов гипервизора. Вместо этого гипервизор использует существующую архитектуру драйверов Windows и обращается к реальным, имеющимся в Windows драйверам устройств. В результате выбора такой архитектуры имеется ряд компонентов, предоставляющих подобное поведение и управляющих этим поведением, которые, в общем и целом, называются стеком гипервизора.

Логически это родительский раздел, отвечающий за предоставление гипервизора, а также за предоставление всего стека гипервизора. Поскольку это компоненты Microsoft, вполне естественно, что корневым разделом может быть только Windows-машина. В собственном пользовании у родительского раздела не должно быть почти никаких ресурсов, поскольку его роль заключается в запуске других операционных систем. Основные компоненты, предоставляемые родительским разделом, показаны на рис. 3.35.

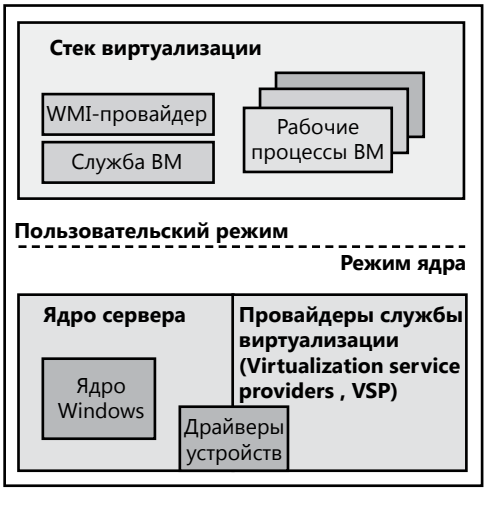


Рис. 3.35. Компоненты родительского раздела

Операционная система родительского раздела

Установка Windows (обычно это минимально возможная серверная установка, которая называется ядром сервера) отвечает за предоставление гипервизора и драйверов устройств для того оборудования, которое установлено в системе (к которому будет обращаться гипервизор), а также за запуск стека гипервизора. Она также является пунктом управления всеми дочерними разделами.

Служба управления виртуальными машинами и рабочие процессы

Служба управления виртуальными машинами (%SystemRoot%\System32\Vmms.exe) отвечает за предоставление гипервизору интерфейса инструментария управления Windows — Windows Management Instrumentation (WMI), — который позволяет управлять дочерними разделами через дополнительный программный модуль консоли управления — Microsoft Management Console (MMC). Она также отвечает за запросы на обмен информацией с приложениями, нуждающимися в связи с гипервизором или с дочерними разделами. Она управляет настройками видимости тех или иных устройств дочерним разделам, управляет порядком распределения памяти и процессорного времени для каждого раздела и выполняет многие другие задачи.

С другой стороны, рабочие процессы виртуальных машин — virtual machine worker processes (VMWP) — выполняют различную виртуализационную работу (подобную работе виртуализационного решения на базе программного обеспечения), которую мог бы выполнять обычный монолитный гипервизор. Это означает управление конечным автоматом (чтобы дать возможность поддержки таких функций, как снимки и переносы состояния) для того или иного дочернего раздела, с несением ответственности за различные уведомления, поступающие от гипервизора, с выполнением эмуляции конкретных устройств, показываемых дочерним разделам, а также совместную работу со службой виртуальных машин и конфигурационными компонентами.

На системе с дочерними разделами, где выполняется множество операций ввода-вывода или привилегированных операций, может ожидаться более высокая степень загруженности центрального процессора, которая должна быть видима в родительском разделе: эти операции можно идентифицировать по имени Vmwp.exe (по одному такому имени для каждого дочернего процесса). Рабочий процесс включает также компоненты, отвечающие за удаленное управление стеком виртуализации, а также компонент RDP, позволяющий использовать удаленный клиентский рабочий стол для подключения к любому дочернему разделу и удаленного просмотра его пользовательского интерфейса и взаимодействия с ним.

Провайдеры службы виртуализации

Провайдеры служб виртуализации — Virtualization service providers (VSP) — отвечают за высокоскоростную эмуляцию конкретных устройств, видимых дочерним разделам[1], и, в отличие от службы виртуальных машин (VM service) и процессов, VSP-провайдеры могут также запускаться как драйверы в режиме ядра. Более подробное описание VSP-провайдеров последует в разделе, посвященном архитектуре устройств в стеке виртуализации.

Драйвер инфраструктуры виртуальных машин и API-библиотека гипервизора

Из-за невозможности непосредственного доступа к гипервизору со стороны приложений пользовательского режима (например, служба виртуальных машин,

[1] Конкретные отличия устройств, эмулированных с помощью VSP и устройств, эмулированных с помощью процессов пользовательского режима, будут рассмотрены чуть позже.

которая отвечает за управление), стек виртуализации должен постоянно обращаться к драйверу в режиме ядра, ответственного за ретрансляцию запросов гипервизору. Этим занимается драйвер инфраструктуры виртуальной машины — VM infrastructure driver (VID). VID также предоставляет поддержку для определенных устройств памяти с малым объемом, таких как эмуляция MMIO и ROM.

Библиотека, находящаяся в режиме ядра, предоставляет гипервизору действующий интерфейс (называемый гипервызовами — hypercalls). Сообщения могут также приходить от дочерних разделов (которые будут выполнять свои гипервызовы), поскольку для всей системы есть только один гипервизор, и он может прислушиваться к сообщениям, поступающим от любого раздела. Эти функциональные возможности можно найти в драйвере устройства Winhv.sys.

Гипервизор

В самой нижней части архитектуры находится гипервизор, который регистрирует себя с процессором в ходе загрузки системы и предоставляет свои службы в пользование стеком (посредством использования интерфейса гипервызовов). Эта начальная инициализация выполняется драйвером hvboot.sys, который настроен на запуск на раннем этапе загрузки системы. Поскольку у процессоров Intel и AMD несколько разная реализация виртуализации с аппаратной поддержкой, фактически имеется два разных гипервизора, из которых с помощью запроса к процессору, посылаемому CPUID-инструкциями в ходе загрузки операционной системы, выбирается один подходящий гипервизор. На Intel-системах загружается двоичный исполняемый файл Hvix64.exe, а на AMD-системах используется образ Hvax64.exe.

Дочерние разделы

Дочерний раздел, как обсуждалось ранее, является экземпляром любой операционной системы, запущенной параллельно родительскому разделу[1]. В отличие от родительского раздела, имеющего полный доступ к APIC, портам ввода-вывода и физической памяти, дочерние разделы ограничены с целью соблюдения мер безопасности и достижения удобства управления своим собственным представлением адресного пространства (гостевым виртуальным адресным пространством — Guest Virtual Address Space, или GVA, — управляемым гипервизором) и отсутствием непосредственного доступа к оборудованию. В сравнении с доступом гипервизора ограничения также сведены в основном к отправке уведомлений и изменениям состояния. Например, у дочернего раздела нет контроля над другими разделами (и нет возможности создавать новые разделы).

У дочерних разделов намного меньше компонентов виртуализации, чем у родительского раздела, поскольку они не несут ответственность за запуск стека виртуализации и отвечают только за связь с ним. Кроме того, эти компоненты могут также считаться необязательными, поскольку они улучшают производи-

[1] Поскольку состояние любого дочернего раздела можно сохранить или ввести работу этого раздела в режим паузы, он не обязательно должен быть работающим, но для него должен быть рабочий процесс.

тельность, но их использование не является критическим. На рис. 3.36 показаны компоненты, которые обычно присутствуют в дочернем разделе Windows.

Рис. 3.36. Компоненты дочернего раздела

ЭКСПЕРИМЕНТ: ИССЛЕДОВАНИЕ ДОЧЕРНИХ РАЗДЕЛОВ ИЗ РОДИТЕЛЬСКОГО РАЗДЕЛА С ПОМОЩЬЮ LIVEKD

Используя средство Sysinternals LiveKd, можно изучить виртуальную машину на основе Windows XP или более поздней версии из родительского раздела, не нуждаясь при этом в перезагрузке дочерней операционной системы в режиме отладки. Сначала при запуске LiveKd нужно указать ключ –hvl, который заставит это средство вывести список идентификаторов и имен активных дочерних разделов:

```
C:\>livekd -hvl

LiveKd v5.0 - Execute kd/windbg on a live system
Sysinternals - www.sysinternals.com
Copyright (C) 2000-2010 Mark Russinovich and Ken Johnson

Partition GUID                          Name
C8FA520B-CBBC-48CE-84EC-14BC2B2C3A74    Win7x64

C:\>
```

Затем нужно запустить LiveKd с ключом –hv, указав идентификатор или имя того дочернего раздела, который нужно исследовать. Как при отладке локальной системы с помощью Livekd, содержимое памяти виртуальной машины можно изменить путем выполнения команд LiveKd, что приведет к тому, что LiveKd видит непостоянства, вызванные данными, отражающими различные моменты времени. Если нужно, чтобы средство LiveKd видело постоянное представление, можно указать ключ –p, чтобы при запуске LiveKd заставить дочерний раздел войти в режим паузы. Все команды, работающие на локальной системе, также работают и при использовании LiveKd для исследования виртуальной машины. Вот как выглядит частичный вывод команды отладчика ядра !vm, которая выводит список различных статистических данных, связанных с памятью, при выполнении дочернего раздела Hyper-V.

```
C:\>livekd -hv win7x64

LiveKd v5.0 - Execute kd/windbg on a live system
Sysinternals - www.sysinternals.com
Copyright (C) 2000-2010 Mark Russinovich and Ken Johnson

Launching C:\program files\Debugging Tools for Windows (x64)\kd.exe:

Microsoft (R) Windows Debugger Version 6.13.0002.895 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.


Loading Dump File [C:\Windows\livekd.dmp]
Kernel Complete Dump File: Full address space is available

Comment: 'LiveKD live system view (hypervisor partition)'
Symbol search path is: srv*c:\Symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows 7 Kernel Version 7600 MP (2 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 7600.16617.amd64fre.win7_gdr.100618-1621
Machine Name:
Kernel base = 0xfffff800`02a06000 PsLoadedModuleList = 0xfffff800`02c43e50
Debug session time: Sat Feb 12 19:34:57.897 17420 (UTC - 7:00)
System Uptime: 3 days 7:14:55.312
Loading Kernel Symbols
...............................................................
...............................................................
........
Loading User Symbols

Loading unloaded module list
......
0: kd> !vm

*** Virtual Memory Usage ***
        Physical Memory:      513422 (   2053688 Kb)
        Page File: \??\C:\pagefile.sys
          Current:   1048576 Kb  Free Space:    792480 Kb
          Minimum:   1048576 Kb  Maximum:      4194304 Kb
        Available Pages:      101260 (    405040 Kb)
        ResAvail Pages:       167196 (    668784 Kb)
        Locked IO Pages:           0 (         0 Kb)
        Free System PTEs:   33533587 ( 134134348 Kb)
        Modified Pages:           898 (      3592 Kb)
```

Клиенты службы виртуализации

Клиенты службы виртуализации — virtualization service clients (VSC-клиенты) — являются в дочернем разделе аналогами провайдеров службы виртуализации (VSP). Как и VSP-провайдеры, VSC-клиенты используются для эмуляции устройств, которая станет темой дальнейших рассмотрений.

Просвещения

Просвещения (enlightenments) являются одной из ключевых оптимизаций производительности, которую использует Windows-виртуализация. Они являются непосредственной модификацией стандартного кода ядра Windows, который способен определить, что данная операционная система запущена в дочернем разделе, и выполнять свою работу по-другому. Обычно эти оптимизации сильно зависят от используемого оборудования, что выражается в гипервызове для уведомления гипервизора. В качестве примера можно привести уведомление гипервизора о продолжительном спин-цикле ожидания занятого ресурса. При таком сценарии у гипервизора может сохраняться некое просроченное состояние вместо отслеживания состояния при каждой отдельно взятой инструкции цикла.

Вход в состояние прерывания и выход из него могут также быть скоординированы с гипервизором, точно так же, как и доступ к APIC, который может быть просвещен с целью предотвращения захвата реального доступа с его последующей виртуализацией.

Другой пример связан с управлением памятью, особенно с TLB-сбросом и изменением адресного пространства (см. главу 9). Обычно операционная система выполняет инструкцию центрального процессора, чтобы произвести сброс этой информации, которая оказывает влияние на весь процессор. Но, поскольку дочерний раздел может совместно использовать центральный процессор со многими другими дочерними разделами, подобная операция может также сбросить такую информацию и для других операционных систем, что приведет к существенным потерям производительности. Если Windows запущена под управлением гипервизора, то она вместо этого выдает гипервызов, заставляя гипервизор сбрасывать только определенную информацию, принадлежащую дочернему разделу.

Эмуляция и поддержка оборудования

Решение вопросов виртуализации может также предоставить оптимизированный доступ к устройствам. К сожалению, большинство устройств не приспособлены для приема сразу нескольких запросов, исходящих от разных операционных систем. Здесь приходится вмешиваться гипервизору, предоставляя по возможности одноуровневую синхронизацию и эмулируя определенные устройства, когда не может быть разрешен реальный доступ к оборудованию. Кроме устройств должны быть также виртуализированы память и процессоры. В табл. 3.26 дается описание трех типов оборудования, которыми может управлять гипервизор.

Таблица 3.26. Виртуализированное оборудование

Компонент	Чем управляется	Использование
Процессор	Встроенным в гипервизор планировщиком и связанными с ним компонентами микроядра	Управляет использованием вычислительной мощностью оборудования, совместно использует несколько процессоров для нескольких дочерних разделов, управляет состоянием процессора (например, регистрами) и переключает его
Память	Встроенным в гипервизор диспетчером памяти и связанными с ним компонентами микроядра	Управляет оперативной памятью оборудования. Защищает память от дочерних разделов и родительского раздела. Предоставляет непрерывное представление физической памяти, начиная с адреса 0
Устройства	Рабочими процессами виртуальной машины, гипервизор отвечает только за перехват и уведомление	Предоставляет множественный доступ к оборудованию, чтобы несколько дочерних процессов могли обращаться к одному и тому же устройству на физической машине. Оптимизирует доступ к физическим устройствам, добиваясь максимально возможной скорости

Вместо того чтобы показывать дочерним разделам реальное оборудование, гипервизор показывает им виртуальные устройства (которые называются VDevs). VDevs скомпонованы как COM-компоненты, которые запускаются внутри рабочего процесса виртуальной машины, и они являются центральным управляемым объектом, стоящим за устройством. Обычно VDevs показывают WMI-интерфейс. Стек виртуализации Windows предоставляет поддержку двух типов виртуальных устройств: эмулированных устройств и синтетических устройств (просвещенный ввод-вывод — enlightened I/O). Для первого из этих типов предоставляется поддержка различных устройств, которые операционные системы в дочерних разделах ожидают найти, а для второго требуется специальная поддержка со стороны гостевой операционной системы. С другой стороны, синтетические устройства предоставляют существенные преимущества с точки зрения повышения производительности, поскольку они сокращают нагрузку на центральный процессор.

Эмулированные устройства

Эмулированные устройства работают за счет предоставления дочернему разделу набора портов ввода-вывода, диапазонов адресов и прерываний, управляемых и отслеживаемых гипервизором. Когда обнаруживается обращение к этим ресурсам, рабочий процесс виртуальной машины в конечном итоге получает уведомление через стек виртуализации (показанный ранее на рис. 3.34). Затем процесс эмулирует ожидаемое от устройства действие и завершает запрос, проходя обратный путь через гипервизор, а затем попадая в дочерний раздел. Только лишь с этой топологической точки зрения можно увидеть, что здесь есть определенные потери производительности, даже если не брать в расчет присущую программной эмуляции аппаратного устройства медлительность.

Потребность в эмулированных устройствах исходит из того факта, что гипервизору нужно поддерживать те операционные системы, которые даже не подозревают о его существовании, а также начальные действия по установке даже самой системы Windows. В ходе загрузки установщик не может просто загрузить все компоненты, требуемые дочерним разделам (например, VSC-клиентов) для использования синтетических устройств, поэтому установка Windows будет всегда использовать эмулированные устройства (поэтому установка кажется очень медленной, но будучи однажды установленной, операционная система будет работать практически со своей обычной скоростью). Эмулированные устройства используются для оборудования, которое не требует высокоскоростной эмуляции и когда программная эмуляция работает быстрее. К таким устройствам можно отнести COM-порты (последовательные порты), параллельные порты или саму материнскую плату.

ПРИМЕЧАНИЕ

Hyper-V эмулирует материнскую плату Intel i440BX, видеокарту S3 Trio и сетевой адаптер Intel 21140 NIC.

Синтетические устройства

Хотя эмулированные устройства работают соразмерно сетевым подключениям со скоростью 10 Мбит/с, VGA-дисплеям низкого разрешения и 16-разрядным звуковым картам, операционные системы и оборудование, обычно используемые дочерними разделами, в наше время требуют намного более высокой вычислительной мощности (например, GbE-подключений со скоростью 1000 Мбит/с, поддержки полноцветной 3D-графики высокого разрешения и высокоскоростного доступа к устройствам хранения информации). Для поддержки такого доступа к виртуализированному оборудованию на приемлемом уровне использования центрального процессора и виртуализированной пропускной способности стек виртуализации использует различные компоненты для оптимизации ввода-вывода устройств до их самых высоких показателей (подобные усовершенствованиям ядра). Эта поддержка складывается из трех компонентов, которые принадлежат тому, что представляется пользователю в виде *компонентов интеграции*, или IC-компонентов:

- провайдеров службы виртуализации (VSP-провайдеров);
- потребителей или клиентов службы виртуализации (VSC-клиентов);
- шины виртуальных машин — VMBus.

Рисунок 3.37 иллюстрирует, как улучшенный ввод-вывод, или ввод-вывод синтетического хранилища, обрабатывается стеком виртуализации.

Как показано на рис. 3.37, VSP-провайдеры работают в родительском разделе, где они связаны с конкретным устройством, за *просвещение* (enlightening) которого они отвечают. При ссылке на синтетические устройства этот термин будет употребляться вместо термина *эмуляция*. VSC-клиенты находятся в дочернем разделе и также связаны с определенным устройством. Но следует заметить, что термин *провайдер* может относиться к нескольким компонентам, разбросанным по всему стеку устройства. Например, в качестве VSP может выступать все нижеперечисленное:

- служба пользовательского режима;
- COM-компонент пользовательского режима;
- драйвер режима ядра.

Во всех трех случаях VSP будет связан с текущим виртуальным устройством внутри рабочего процесса виртуальной машины. С другой стороны, VSC-клиенты почти всегда разрабатываются в качестве драйверов, находящихся на самом нижнем уровне стека, и перехватывают операции ввода-вывода, относящиеся к устройству, перенаправляя их по более оптимальному пути. Основной оптимизацией, выполняемой данной моделью, является исключение реального обращения к устройству и использование вместо этого шины VMBus. Согласно данной модели гипервизор ничего не знает о вводе-выводе, и VSP перенаправляет этот ввод-вывод непосредственно в стек хранилища ядра родительского раздела, исключая к тому же еще и переход в пользовательский режим. Другие VSP-провайдеры могут выполнять работу непосредственно на устройстве, общаясь с реальным оборудованием и обходя любой драйвер, который мог быть

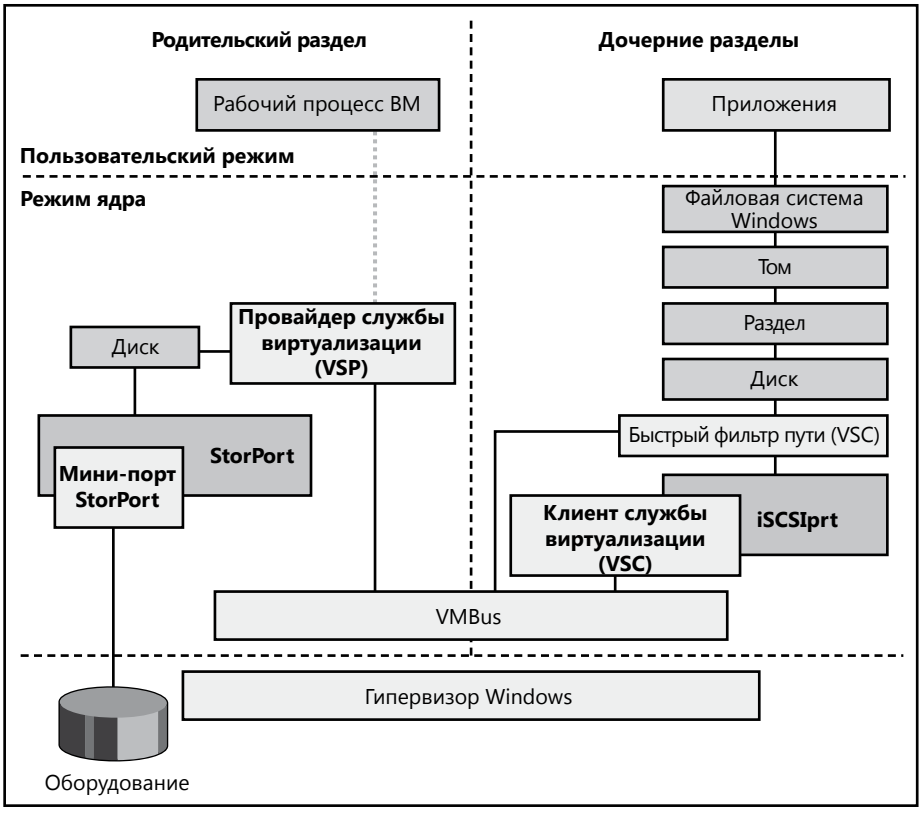


Рис. 3.37. Части Hyper-V, обрабатывающие ввод-вывод

загружен в родительском разделе. Еще один вариант предполагает наличие VSP-провайдера пользовательского режима, надобность в котором может появиться при работе с устройствами, имеющими низкую полосу пропускания.

Ранее уже говорилось, что VMBus — это название шинного транспорта, использованного для оптимизации обращения к устройству путем реализации протокола связи, использующего службы гипервизора. VMBus является драйвером шины, который имеется как в родительском, так и в дочерних разделах и отвечает за учет «Plug and Play» синтетических устройств в дочернем разделе. В нем также содержится оптимизированный протокол обмена сообщениями между разделами, который использует метод транспортировки, соответствующий размеру данных. Один из таких методов заключается в предоставлении общего кольцевого буфера между каждым разделом. Этот буфер, по существу, является областью памяти, в которую загружается определенное количество данных с одной стороны, и выгружается с другой стороны. Здесь не нужно выделять или освобождать какое-нибудь пространство памяти, потому что буфер постоянно используется снова и снова и просто подвергается циклическому сдвигу. Со временем он может наполниться запросами, это будет означать, что более новые данные ввода-вывода будут переписывать более старые данные ввода-вывода. При таком довольно редком развитии событий VMBus просто придержит самые новые запросы, пока

не будут завершены самые старые запросы. Другая разновидность передачи сообщений заключается в непосредственном отображении дочерней памяти на родительское адресное пространство для достаточно больших передач.

Виртуальные процессоры

Так же как гипервизор не допускает прямого обращения к оборудованию (или к памяти, как будет показано далее), дочерний процесс не видит настоящих процессоров, имеющихся на машине, но имеет виртуальное представление о центральных процессорах. На корневой машине администратор и операционная система имеют дело с *логическими процессорами*, являющимися настоящими процессорами. На этих процессорах могут быть запущены потоки (например, двухпроцессорная машина с процессорами, состоящими из четырех ядер, имеет восемь логических процессоров), и назначают эти процессоры различным дочерним разделам. Например, для работы одного дочернего раздела могут быть спланированы логические процессоры 1, 2, 3 и 4, а для работы второго дочернего раздела могут быть спланированы процессоры 5, 6, 7 и 8. Возможность всех этих операций обусловлена использованием *виртуальных процессоров*, или VP.

Поскольку процессоры могут совместно использоваться сразу несколькими дочерними разделами, в гипервизор включен свой собственный планировщик, распределяющий рабочую нагрузку различных разделов на каждом процессоре. Кроме того, гипервизор обслуживает состояние регистров каждого виртуального процессора, и каждого соответствующего «переключения процессора», когда один и тот же логический процессор используется другим дочерним разделом. Родительский раздел имеет возможность доступа ко всем этим контекстам и внесения в них изменений по мере необходимости и является существенной частью стека виртуализации, который должен отвечать за конкретные инструкции и выполняемые действия.

Гипервизор также непосредственно отвечает за виртуализированные APIC-контроллеры процессора и за предоставление простого, менее функционального виртуального APIC, включая поддержку таймера, которая находится в большинстве APIC-контроллеров (но при более медленной частоте). Поскольку APIC-контроллеры поддерживаются не всеми операционными системами, гипервизор также допускает вставку прерываний посредством гипервызовов, что позволяет стеку виртуализации эмулировать стандартный PIC-контроллер i8059.

И наконец, поскольку Windows поддерживает динамическое добавление процессоров, администратор может в ходе работы добавлять новые процессоры к дочернему разделу для повышения быстроты реагирования сильно загруженных гостевых операционных систем.

Виртуализация памяти

Заключительной частью оборудования, которое должно быть абстрагировано от дочерних разделов, является память — не только для нормального поведения гостевых операционных систем, но также для обеспечения безопасности и стабильности. Неправильное управление доступом к памяти со стороны дочернего раздела может привести к разглашению закрытой информации и повреждению данных, а также может позволить вредоносные атаки за счет «побега» из

дочернего раздела и нападения на родительский раздел (что позволит затем атаковать другие дочерние разделы). Помимо этого аспекта встает также вопрос представления гостевой операционной системой физического адресного пространства. Почти все операционные системы ожидают, что память начнется с адреса 0 и будет представлять собой некое непрерывное пространство, поэтому простое назначение участков физической памяти каждому дочернему разделу не будет работать, даже если на системе будет достаточный объем доступной памяти.

Для решения этой проблемы гипервизор реализует адресное пространство, называемое *гостевым физическим адресным пространством* (guest physical address space, GPA-пространство). GPA начинается с адреса 0, что удовлетворяет потребности операционных систем внутри дочерних разделов. Но GPA не является простым отображением на участок физической памяти из-за наличия второй проблемы — дефицита непрерывной памяти. Гостевое физическое адресное пространство как таковое может указать на любое место физической памяти машины (*системное физическое адресное пространство* — system physical address space, или SPA-пространство), и необходима система преобразования для перехода от одного типа адресов к другому. Эта система преобразования поддерживается гипервизором, и она почти идентична тому способу, с помощью которого виртуальная память отображается на физическую память на процессорах x86 и x64.

Что касается действительных виртуальных адресов в дочернем разделе (*гостевое виртуальное адресное пространство*, GVA-пространство), они продолжаются таким образом, чтобы быть управляемыми операционной системой без какого-либо изменения в поведении. Операционная система верит в то, что реальные физические адреса, которые находятся в ее собственных таблицах страниц, на самом деле действительно принадлежат SPA-пространству. На рис. 3.38 дано общее представление об отображении между каждым из уровней.

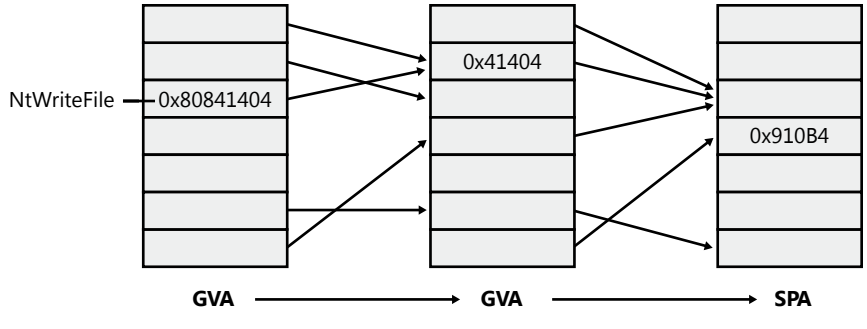


Рис. 3.38. Преобразование гостевых виртуальных и физических адресов

Это означает, что когда гостевая операционная система загружается и создает таблицы страниц для отображения виртуальной памяти на физическую, гипервизор перехватывает SPA-адреса и сохраняет свою собственную копию таблиц страниц. Согласно используемой концепции, когда фрагмент кода обращается по виртуальному адресу внутри гостевой операционной системы, гипервизор осуществляет начальное преобразование таблицы страниц для перехода от гостевого виртуального адреса к GPA, а затем отображает этот GPA-адрес на соот-

ветствующий SPA-адрес. В действительности эта операция оптимизирована за счет использования теневых таблиц страниц (shadow page tables, SPT-таблицы), которые поддерживаются гипервизором, чтобы получить непосредственные преобразования GVA-адресов в SPA-адреса и производить простую загрузку, если такое возможно, чтобы гостевые операционные системы обращались по SPA-адресам напрямую.

ПРЕОБРАЗОВАНИЕ АДРЕСОВ ВТОРОГО УРОВНЯ И ТЕГИРОВАННЫЙ БУФЕР БЫСТРОГО ПРЕОБРАЗОВАНИЯ АДРЕСА (TLB)

Поскольку преобразование из GVA в GPA и SPA является весьма затратным делом (поскольку должно осуществляться программно), производители центральных процессоров работали над тем, чтобы уменьшить эту неэффективность, сделав такой процессор, который изначально будет знать о требованиях трансляции адресов виртуальной машины. Иными словами, усовершенствованный процессор мог понять, что обращение к памяти осуществляется из находящейся на хосте виртуальной машины и самостоятельно провести поиск соответствия GVA и SPA, не требуя для этого участия гипервизора. Эта технология поиска называется преобразованием адреса второго уровня — Second-Level Address Translation (SLAT), — поскольку она охватывает как преобразование от цели к хосту (второй уровень), так и преобразование виртуального адреса хоста в физический адрес хоста (первый уровень). Но, руководствуясь рыночными интересами, компания Intel назвала эту поддержку технологией VT Extended/Nested Page Table (NPT), а компания AMD назвала ее AMD-V Rapid Virtualization Indexing (RVI).

Самая последняя версия стека Hyper-V полностью использует преимущества этой поддержки со стороны процессоров, упрощая свой код и минимизируя количество контекстных переключений, требуемых для обработки ошибок обращения к страницам в разделах, размещенных на хост-машинах. Кроме того, SLAT позволяет Hyper-V отказаться от имеющихся в этом стеке теневых таблиц страниц и относящегося к ним отображения, что позволяет еще больше сократить издержки обращений к памяти. Эти изменения улучшили масштабируемость Hyper-V на таких системах, в особенности это привело к увеличению максимального количества виртуальных машин, обслуживаемых одним хостом (сервером Hyper-V) или запускаемых в параллельном режиме. В соответствии с тестами, выполненными компанией Microsoft, поддержка SLAT увеличивает максимальное количество поддерживаемых сеансов в 1,6–2,5 раза. Кроме того, издержки процессора упали с 10 до 2 %, и каждая виртуальная машина занимает на хосте на один мегабайт физической оперативной памяти меньше.

Вдобавок, как Intel, так и AMD предоставили функциональные возможности, которые, как правило, встречались только на таких RISC-процессорах, как ARM, MIPS или PPC. Имеется в виду способность процессора различать процессы, связанные с каждой кэшированной записью преобразования виртуального адреса в физический в буфере быстрого преобразования адреса — translation look-aside buffer (TLB). На таких CISC-процессорах, как x86 и x64, буфер TLB был создан как общесистемный ресурс — при каждом переключении операционной системой текущего выполняемого процесса буфер TLB должен был быть очищен, чтобы аннулировать любые кэшированные записи, которые могли бы принадлежать предыдущему выполняемому процессу. Если процессору вместо этого можно было бы сообщить, что про-

цесс изменился, буфер TLB мог бы избежать очистки, и процессор мог бы просто не использовать кэшированные записи, не соответствующие этому процессу. Могли бы создаваться новые записи, которые со временем переписывали бы старые. Такой тип более рационального буфера TLB называется тегированным TLB, поскольку каждая запись в кэш-памяти тегирована идентификатором, принадлежащим тому или иному процессу.

Очистка TLB еще хуже, чем работа с системами Hyper-V, поскольку различные процессы могут на самом деле относиться к совершенно различным виртуальным машинам. Иными словами, всякий раз, когда гипервизор и операционная система планируют работу другой виртуальной машины, TLB хоста должен быть очищен с удалением всех кэшированных преобразований предыдущей виртуальной машины, замедляя при этом доступ к памяти и вызывая существенные задержки. При запуске на процессоре с реализованным тегированным TLB Hyper-V может просто уведомит процессор, что запущен новый процесс или виртуальная машина и что теперь нужно использовать записи другой виртуальной машины. Процессоры AMD с RVI поддерживают тегированные TLB-буферы через идентификатор адресного пространства — Address Space Identifier, или ASID, а недавно появившиеся процессоры Intel Nehalem-EX processors реализуют тегированные TLB путем использования идентификатора виртуального процессора — Virtual Processor Identifier (VPID).

ДИНАМИЧЕСКАЯ ПАМЯТЬ

Свойство, называемое динамической памятью (Dynamic Memory), позволяет системным администраторам создать переменную выделения виртуальным машинам физической памяти на основе потребностей в памяти имеющихся виртуальных машин, практически таким же способом, с помощью которого диспетчер памяти Windows подгоняет объем физической памяти, выделяемой каждому процессу на основе его потребностей в памяти. Такая возможность означает, что администраторы не должны с высокой точностью оценивать размер виртуальной машины, требуемый для оптимальной производительности, и что системная физическая память используется теми виртуальными машинами, которые в ней нуждаются более эффективно.

Архитектура динамической памяти состоит из нескольких компонентов, показанных на рис. 3.39.

Основными компонентами архитектуры являются:

- Балансировщик динамической памяти, реализованный в службе управления виртуальными машинами. Балансировщик отвечает за выделение физической памяти дочерним разделам.
- VSP динамической памяти (DM VSP), который запускается в рабочих процессах виртуальных машин (VMWP) дочерних разделов, имеющих включенный режим динамической памяти.
- VSC динамической памяти (DM VSC, %SystemRoot%\System32\Drivers\Dmvsc.sys), установленный в виде драйвера просвещения (enlightenment driver), который запущен в дочерних разделах.

Для настройки виртуальной машины на использование динамической памяти администратор выбирает в настройках памяти виртуальной машины позицию переключателя Dynamic (Динамическая) (рис. 3.40).

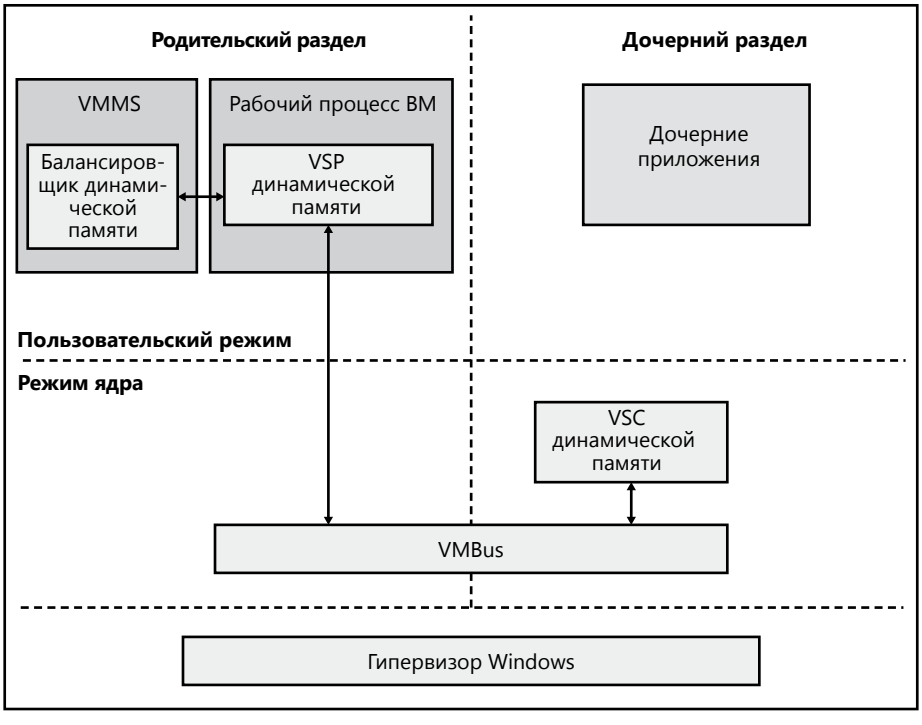


Рис. 3.39. Архитектура динамической памяти

Связанные настройки включают объем оперативной памяти, который будет выделен виртуальной машине при ее запуске (Startup RAM), максимальный объем, который ей может быть выделен (Maximum RAM), процентный показатель объема памяти виртуальной машины, который будет доступен для немедленного использования операционной системой при повышении ее потребностей в памяти, и наконец, меру значимости виртуальной машины по отношению к другим виртуальным машинам. Помимо того что эти настройки служат мерой распределения физической памяти среди виртуальных машин с включенным режимом динамической памяти, гипервизор также использует их в качестве руководства для порядка запуска виртуальных машин, настроенных на запуск при загрузке системы. И наконец, процентный показатель доступной памяти является ссылкой на память внутри виртуальной машины, которую операционная система виртуальной машины не выделяет процессу, драйверам устройств или самой себе и которая может быть выделена без опасения за возникновение ошибки обращения к странице.

При запуске DM VSC в дочернем разделе, у которого в настройках памяти включен режим динамической памяти, сначала производится проверка, поддерживает ли операционная система возможности динамического распределения памяти. Эта проверка проводится путем простого вызова функции горячего добавления памяти в диспетчере памяти, с указанием блока дочерней физической памяти, уже выделенного виртуальной машине. Если диспетчер памяти поддерживает горячее добавление, он возвращает ошибку, свидетельствующую о том, что диапазон адресов уже используется,

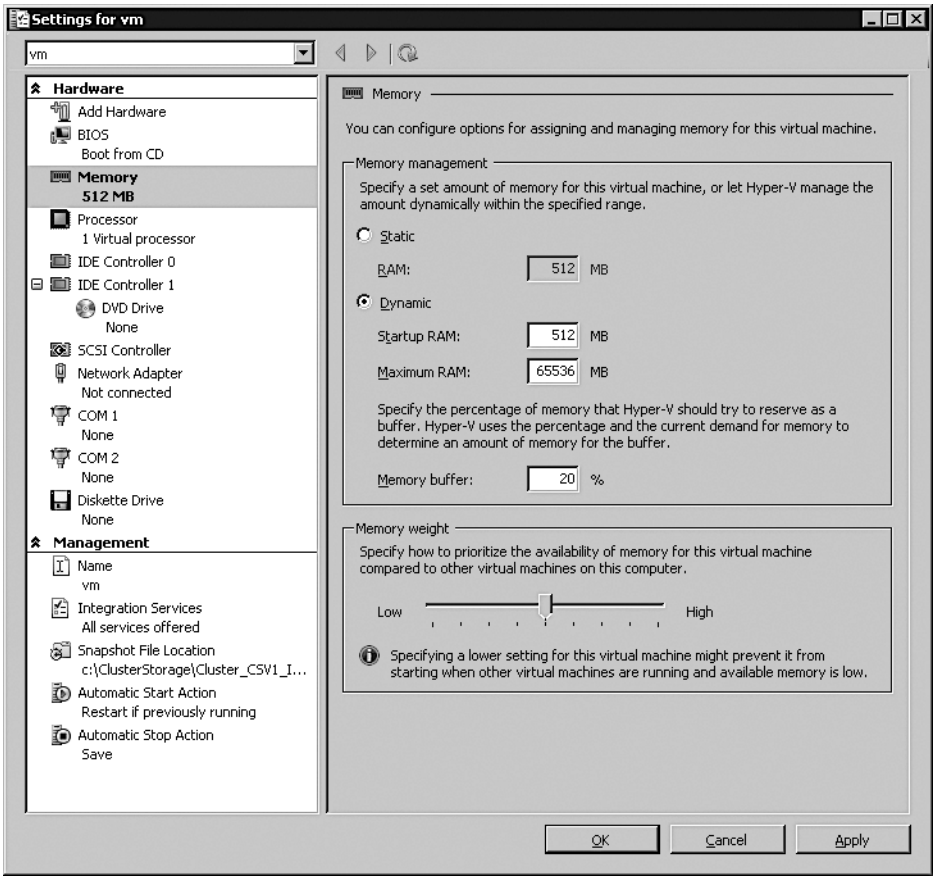


Рис. 3.40. Диалоговое окно настройки динамической памяти

а если он такое добавление не поддерживает, то он сообщает, что такая функция не поддерживается. Если динамическая память поддерживается, DM VSC устанавливает подключение к DM VSP через VMBus. Поскольку использование системной памяти меняется в ходе процесса загрузки, после того как все запускаемые в автоматическом режиме службы Windows завершат инициализацию, VSC начинает один раз в секунду выдавать статистические показатели памяти, показывающие имеющийся в виртуальной машине текущий уровень выделяемых системных ресурсов.

DM VSP в родительском разделе вычисляет объем необходимой памяти для соответствующих этому разделу виртуальных машин, используя следующую формулу на основе отчета о памяти виртуальной машины:

Потребность в памяти = Выделенная память / Физическая память

Значение *Физическая память* ссылается на объем памяти, уже выделенной разделу виртуальной машины. Также отслеживается показатель непрерывной экспоненциальной средней потребности, представляющего собой 20 секунд предыдущих отчетов о потребностях, регулируя среднюю потребность только в том случае, когда текущая потребность отклоняется от средней по крайней мере на величину стандартного отклонения.

Компонент, который называется балансировщиком, работает в VMMS-службе. Один раз в секунду он анализирует потребности в памяти, о которых заявляют провайдеры DM VSP, учитывает политику конфигурации виртуальной машины и определяет, может ли быть перераспределен больший объем памяти и каким образом это может быть сделано. Если включены глобальные настройки Hyper-V под названием *NUMA spanning*, балансировщик использует два балансировочных механизма: один механизм является глобальным балансировщиком, отвечающим за назначение новых виртуальных машин NUMA-узлам. Он выполняет свою работу на основе данных об использовании памяти и потребностей виртуальной машины в узлах на время назначения. У каждого NUMA-узла есть свой собственный локальный балансировщик, управляющий распределением памяти узла между виртуальными машинами, назначенными этому узлу. Если режим NUMA spanning отключен, глобальному балансировщику ничего не остается делать, кроме пробуждения для системы только локального балансировщика.

Преимущество назначения виртуальных машин NUMA-узлам состоит в том, что виртуальным машинам будет гарантироваться наиболее быстрый доступ к памяти из всех возможных вариантов. А недостаток заключается в том, что возможности запуска виртуальной машины или добавления ей памяти может и не быть, если в сумме нераспределенной памяти хватает, но ни у одного узла не имеется в достаточном количестве доступной памяти, чтобы обеспечить требуемый объем.

Локальный балансировщик увеличивает или уменьшает глобальные целевые потребности в памяти, чтобы использовать всю доступную память под своим управлением или чтобы использовать ее до тех пор, пока не будет достигнут минимальный уровень потребности, показывающий, что у всех виртуальных машин имеется вполне достаточный объем памяти. Затем балансировщик проводит циклический обход всех виртуальных машин, определяя, сколько памяти добавить или убрать из каждой виртуальной машины, чтобы достичь целевой потребности. В ходе вычислений балансировщик резервирует для хоста минимальный объем памяти. Базовый объем резервируемой для хоста памяти составляет примерно 400 Мбайт плюс 30 Мбайт для каждого 1 Гбайт оперативной памяти системы. Факторы, которые могут повлиять на объем резервируемой памяти, включают в себя такие обстоятельства, использует ли система SLAT, или программную организацию страничной памяти, а также включено ли мультимедийное перенаправление. Каждые пять минут балансировщик также удаляет память из тех виртуальных машин, у который ее настолько много, что их потребности практически равны нулю.

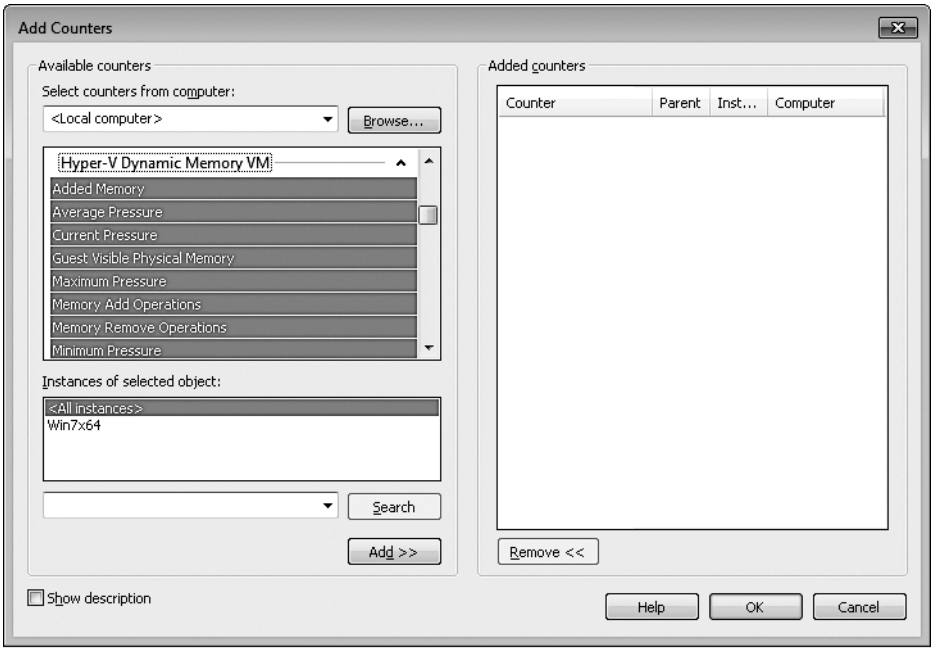
Следует учесть, что если запущенная операционная система дочернего раздела является 32-разрядной версией Windows, механизм динамической памяти не станет назначать раздел, превышающий по объему 4 Гбайт памяти.

Как только будет вычислен объем добавляемой или удаляемой из виртуальных машин памяти, балансировщик просит каждый рабочий процесс (WP) выполнить нужную операцию. Если эта операция должна удалить память, WP сигнализирует дочернему DM VSC через VMBUS об удаляемом объеме, и DM VSC увеличивает объем своей используемой памяти (создает «пузырь») за счет выделения физической памяти из системы, используя функцию MmAllocatePagesForMdlEx. Она извлекает выделенные GPA-адреса и отправляет назад рабочему процессу, который передает их диспетчеру памяти Hyper-V. Тот, в свою очередь, проводит преобразование GPA-адресов в SPA-адреса и добавляет память к своему свободному пулу памяти.

Если операция касается добавления памяти, WP запрашивает сначала у диспетчера памяти Hyper-V, есть ли у виртуальной машины какая-нибудь выделенная ей физическая память, но на данный момент распределенная в качестве «пузыря» VSC. Если такая память есть, WP извлекает GPA-адреса для объема, который должен быть взят у «пузыря», и просит VSC освободить эти страницы, делая их опять доступными для использования операционной системой виртуальной машины. Если объем, который может быть освобожден за счет ликвидации «пузыря», оказывается меньше того объема физической памяти, который балансировщик хочет дать виртуальной машине, он просит диспетчер памяти Hyper-V выделить оставшийся объем из его пула свободной памяти дочернему разделу через поддержку со стороны Windows горячего добавления памяти и рапортует рабочему процессу о добавленных GPA-адресах, а тот, в свою очередь, переправляет их дочернему DM VSC.

ЭКСПЕРИМЕНТ: НАБЛЮДЕНИЕ ЗА ДИНАМИЧЕСКОЙ ПАМЯТЬЮ

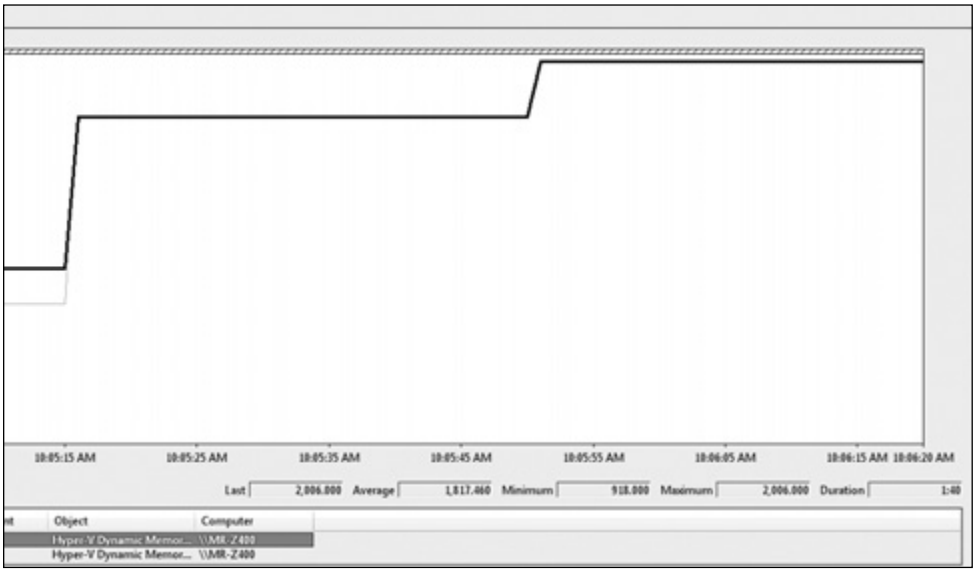
Поведение динамической памяти можно пронаблюдать, настроив этот режим памяти для виртуальной машины, запущенной на совместимой с динамической памятью 64-разрядной операционной системой, такой как Windows 7 или Windows Server 2008 R2. Hyper-V выставляет несколько счетчиков производительности, связанных с динамической памятью под Hyper-V Dynamic Memory Balancer и Dynamic Memory VM. Счетчики включают объем памяти, выделенной гостевой операционной системе — видимой памяти гостевой операционной системы — (тот объем, который с ее точки зрения у нее имеется), ее текущие и средние потребности в памяти, и объем памяти, добавленный и удаленный в течение определенного времени:



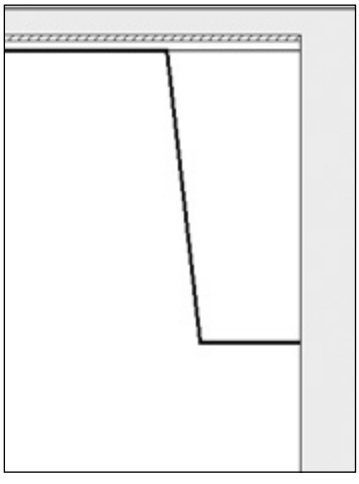
После свежей загрузки виртуальной машины добавьте счетчики Guest Visible Physical Memory и Physical Memory. Установите масштаб, превышающий в три раза текущее значение гостевой видимой физической памяти — Guest Visible Physical Memory value, которое будет как минимум таким же большим, как значение физической памяти — Physical Memory. Затем запустите в виртуальной машине инструментальную программу Sysinternals Testlimit со следующей командной строкой:

```
testlimit -m 1000 -c 1
```

Если предположить наличие достаточного количества физической памяти на вашей системе, это заставит Testlimit выделить около 1 Гбайт виртуальной памяти, подняв потребности в памяти на виртуальной машине. Через несколько секунд вы увидите, что гостевая видимая и текущая физическая память, выделенная виртуальной машине, достигли одинаковых значений. Еще примерно через 30 секунд вы увидите еще один скачок, когда балансировщик решит, что дополнительной памяти недостаточно для полного удовлетворения потребностей в памяти на виртуальной машине, и, поскольку у хоста есть еще доступная память, даст виртуальной машине еще немного памяти.



Если завершить работу программы Testlimit, уровни памяти останутся постоянными в течение нескольких минут, если не будет никаких потребностей в памяти от хоста или от других виртуальных машин, но в конечном итоге балансировщик ответит на отсутствие в потребностях памяти на виртуальной машине путем подгонки памяти. Обратите внимание на то, что показания счетчика Guest Visible Physical Memory остается неизменными, в то время как показания счетчика Physical Memory падают, возвращаясь примерно на тот же уровень, который был до выполнения программы Testlimit. ■



Перехваты

Мы говорили о различных способах виртуализации доступа к оборудованию, процессорам и памяти за счет использования гипервизора, а иногда и за счет передачи управления рабочему процессу виртуальной машины, но мы еще не говорили о том механизме, который позволяет всему этому произойти — о перехватах. Перехваты — это настраиваемые ловушки, которые родительский раздел может установить и настроить в целях реагирования на те или иные обстоятельства. В их число могут входить:

- Перехваты ввода-вывода, используемые для эмуляции устройств.
- MSR-перехваты, используемые для эмуляции и профилирования APIC.
- Обращение к GPA-адресам, используемое для эмуляции устройств, мониторинга и профилирования. (Кроме того, перехват может быть тонко настроен на определенное обращение, например на чтение, запись или выполнение.)
- Перехваты исключений, например ошибок обращений к страницам, используемые для сохранения состояния машины и эмуляции памяти (например, обслуживание копирования при записи).

Как только гипервизор определяет событие, для которого был зарегистрирован перехват, он отправляет сообщение о перехвате через стек виртуализации и вводит виртуальный процессор в приостановленное состояние. Стек виртуализации (обычно рабочий процесс) должен обработать событие и возобновить работу виртуального процессора (обычно путем изменения регистра состояния, отражающего работу, выполняемую для обработки перехвата).

Динамическая миграция

Для поддержки таких сценариев, как плановое обновление оборудования, балансирование нагрузки на ресурсы между серверами в Hyper-V включена поддержка миграций виртуальных машин между узлами отказоустойчивого кластера Windows — Windows Failover Cluster — с минимальными простоями. Ключом

эффективности для динамической миграции (Live Migration) служит то, что основная часть переноса памяти виртуальной машины от источника к приемнику происходит в тот момент, когда виртуальная машина продолжает работать на исходном узле. Виртуальная машина приостанавливает работу, а потом возобновляет ее уже на целевом узле, но только тогда, когда завершится перенос памяти. В это малое время (когда мигрируется финальное состояние виртуальной машины), которое обычно меньше, чем значение по умолчанию лимита времени TCP, сохраняется открытое подключение от клиентов с использованием служб виртуальной машины, и миграция с их точки зрения делается прозрачной.

Процесс динамической миграции происходит в несколько этапов, показанных на рис. 3.41:

1. **Настройка миграции.** VMMS-служба ведущего (исходного) узла виртуальной машины открывает TCP-подключение к узлу назначения. Она переносит на узел назначения информацию о конфигурации виртуальной машины, которая включает такие спецификации виртуального оборудования, как количество процессоров и объем оперативной памяти. VMMS-служба на конечном (целевом) узле создает экземпляр остановленной виртуальной машины, соответствующей конфигурации. VMMS-служба уведомляет рабочий процесс виртуальной машины о том, что динамическая миграция готова к продолжению, и передает ему TCP-подключение. В то же время VMMS-служба целевого узла передает свою сторону подключения рабочему процессу этого узла.

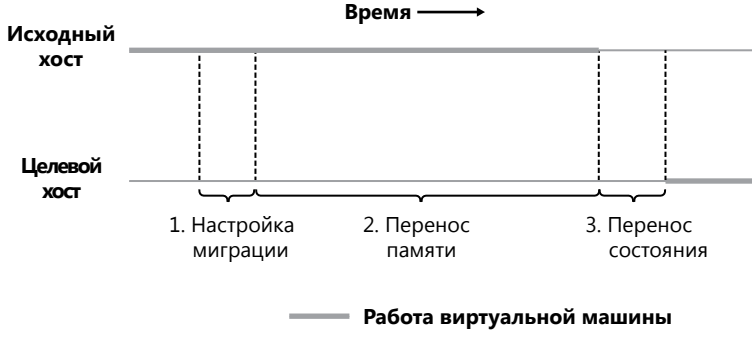


Рис. 3.41. Этапы переноса виртуальной машины во время динамической миграции

2. **Перенос памяти.** Этап переноса памяти состоит из нескольких подэтапов:
   1) Исходный VMWP создает битовый массив, в котором один бит представляет каждую страницу физической памяти гостевой операционной системы виртуальной машины. Он устанавливает каждый бит, чтобы показать, что страница находится в состоянии необработанности (dirty), это означает, что текущее содержимое страницы еще не было отправлено на целевой узел.
   2) VMWP источника регистрирует функцию обратного вызова уведомления об изменении памяти наряду с тем, что гипервизор устанавливает соответствующий бит в битовом массиве для каждой страницы виртуальной машины, подвергающейся изменениям.

3) VMWP источника продолжает обход битового массива dirty-страниц, соответствующего 16-килобайтным блокам, очищая dirty-биты в этом массиве для страниц блока, считывая через вызов гипервизора содержимое каждой dirty-страницы и отправляя содержимое на целевой узел. VMWP приемника вызывает гипервизор для вставки содержимого памяти в физическую память гостевой операционной системы целевой виртуальной машины.
4) По окончании перебора битового массива dirty-страниц VMWP источника проводит проверку, не была ли какая-либо из страниц изменена (переведена в разряд dirty) в ходе этого перебора. Если нет, происходит переход к следующему этапу миграции, но если какая-нибудь страница была изменена, перебор повторяется. Если перебор битового массива прошел пять раз, значит, виртуальная машина вносит изменения быстрее, чем рабочий процесс способен отправлять эти изменения, поэтому он приступает к следующему этапу миграции.

3. **Перенос состояния.** VMWP источника приостанавливает работу виртуальной машины и проводит завершающий перебор битового массива dirty-страниц, чтобы переслать любые страницы, которые подверглись изменению за время последнего перебора. Поскольку на время переноса виртуальная машина была приостановлена, страницы больше изменяться не будут. Затем рабочий процесс источника отправляет состояние виртуальной машины, включая содержимое регистров виртуального процессора. И наконец, он уведомляет VMMS-службу о том, что миграция завершена, ждет подтверждения о получении этого уведомления, а затем отправляет сообщение на целевой узел, передавая ему владение виртуальной машиной. В качестве последнего этапа миграции рабочий процесс приемника переводит виртуальную машину в рабочее состояние.

Еще одним аспектом динамической миграции является передача владения файлами виртуальной машины, включая ее виртуальные жесткие диски (VHD). Традиционная Windows-кластеризация — Traditional Windows Clustering — является моделью, в которой отсутствует какое-либо совместное использование, где каждый логический номер устройства (LUN) кластерной системы хранения данных в каждый момент времени является собственностью только одного узла. Узел, владеющий LUN, имеет исключительный доступ к LUN-устройству и к любому, хранящемуся на нем файлу. Эта модель может привести к усложнению управления, поскольку каждая виртуальная машина должна храниться на отдельном LUN-устройстве, а значит, на отдельном томе, что приведет к взрывному росту томов на кластере, предоставляющем хостинг для множества виртуальных машин. Это создает еще более серьезную проблему для динамической миграции, поскольку владение LUN-устройством передается в ресурсоемкой операции, состоящей из того, что исходный узел смещает любые измененные файловые данные в LUN, исходный узел демонтирует тома, отформатированные на LUN, владение передается из исходного узла целевому узлу, и целевой узел монтирует тома. В зависимости от количества томов на LUN-устройстве и количества измененных данных, которые должны быть опять записаны, вся последовательность может занять несколько десятков секунд, что может помешать динамической миграции достичь ее цели быть воспринятой в качестве почти мгновенной миграции.

Для устранения ограничений, накладываемых традиционной моделью кластеризации и предоставления возможности проведения динамической миграции, эта миграция использует функцию хранения под названием «кластерные общие тома» — Clustered Shared Volumes (CSV). Используя CSV, один узел владеет пространством имен томов на LUN, в то время как все остальные могут иметь исключительные права владения на отдельные файлы. Исключительные права владения позволяют узлу, предоставляющему хостинг для виртуальной машины, непосредственно обращаться к хранилищу на диске, содержащему VHD-файл, обходя тем самым обращения через сетевую файловую систему, которые обычно требуются для взаимодействия с LUN-устройством, находящимся во владении другого узла. Только когда узел желает создать или удалить файлы, изменить размер файлов (например, увеличить размер динамического или дифференцированного VHD) или изменить другие метаданные файла, например отметку времени, ему нужно отправить запрос через протокол SMB2 узлу-владельцу, если он таковым не является.

Гибридная модель совместного использования CSV позволяет владельцу LUN оставаться без изменений в ходе динамической миграции и позволяет только владельцам отдельных файлов мигрирующих виртуальных машин вносить изменения, избегая операций демонтирования и монтирования. Кроме того, измененные данные, характерные для файлов виртуальной машины, должны быть записаны перед миграцией, и это должно происходить параллельно с миграцией памяти. На рис. 3.42 изображено изменение прав владения в ходе динамической миграции.

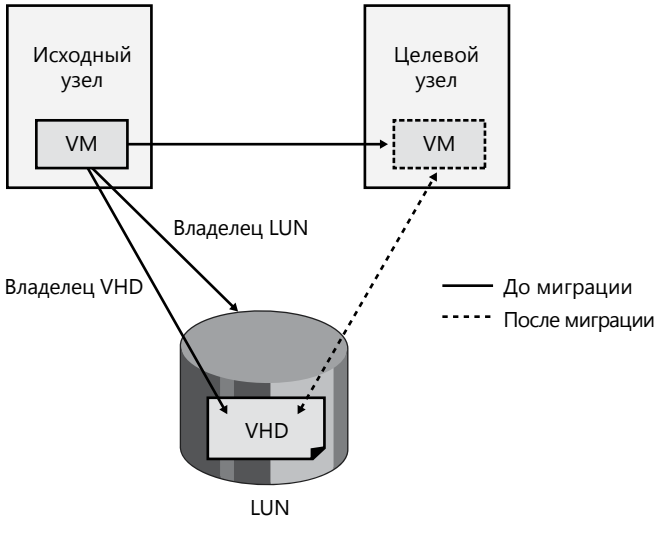


Рис. 3.42. Кластерные общие тома в динамической миграции

Диспетчер транзакций ядра

Одним из наиболее трудоемких аспектов разработки программного обеспечения являются условия обработки ошибок. Особенно актуальным это утверждение становится в том случае, когда при выполнении операции высокого уровня

приложение завершило одну или несколько подзадач, приведших к изменениям в файловой системе или реестре. Например, прикладная программа, обновляющая службу, могла произвести несколько обновлений реестра, переместить один из исполняемых файлов приложения, а затем ей было отказано в доступе при попытке обновить второй исполняемый файл. Если служба не хочет оставить приложение в сложившемся противоречивом состоянии, она должна отследить все внесенные им изменения и приготовиться к их отмене. Тестирование кода восстановления при возникновении ошибок представляет определенные трудности и поэтому часто пропускается, и ошибки в коде восстановления могут свести на нет все усилия.

Приложения могут при минимуме усилий получить автоматические возможности восстановления в случае возникновения ошибок путем использования механизма ядра под названием диспетчера транзакций ядра — Kernel Transaction Manager (KTM), — предоставляющего средства, требуемые для выполнения таких транзакций и допускающего использование в пользовательском режиме таких служб, как координатор распределенных транзакций — distributed transaction coordinator (DTC). Этими службами могут воспользоваться любые разработчики, использующие соответствующие API-функции.

KTM разрешает не только весьма масштабные проблемы, подобные вышеописанной. Даже на домашнем компьютере отдельно взятого пользователя установка исправлений службы или осуществление восстановления системы являются масштабными операциями, вовлекающими как файлы, так и разделы реестра. Отключение более старых Windows-компьютеров от электропитания в ходе подобных операций оставляло весьма призрачные надежды на успешную загрузку. Даже при том, что файловая система NT File System (NTFS) всегда имела файл журнала, позволяющий ей гарантировать проведение атомарных операций, это лишь означало, что тот файл, который записывался в ходе данного процесса, был бы полностью записан или полностью удален. Это не гарантировало полноценное проведение операций обновления или восстановления. Кроме того, с годами появились многие усовершенствования и в реестре, позволяющие разбираться с повреждениями (см. главу 4), но исправления применяются только на уровне раздела и параметра.

С целью поддержки транзакции KTM позволяет программам, управляющим ресурсами, подвергающимся транзакции, таким как NTFS и реестру, координировать их обновления для конкретного набора изменений, внесенных приложением. В NTFS для поддержки транзакций используется расширение, которое называется TxF. В реестре используется похожее расширение, которое называется TxR. Эти диспетчеры ресурсов режима ядра работают совместно с KTM для координации состояния транзакции, подобно тому, как диспетчеры ресурсов пользовательского режима используют DTC для координации состояния транзакции между несколькими диспетчерами ресурсов пользовательского режима. KTM может также использоваться третьими сторонами для реализации их собственных диспетчеров ресурсов.

И TxF, и TxR определяют новый набор API-функций файловой системы и реестра, которые похожи на уже существующие функции, за исключением того, что в них включаются параметры транзакции. Если приложению нужно создать файл в рамках транзакции, оно сначала использует KTM для создания транзакции,

а затем передает дескриптор получившейся транзакции API-функции, создающей новый файл. Хотя реализации KTM в реестре и NTFS будут рассмотрены позже, это не является их единственным возможным применением. На самом деле предоставляются четыре системных объекта, позволяющих поддерживать множество операций. Эти операции перечислены в табл. 3.27.

Таблица 3.27. Объекты KTM

Объект	Значение	Использование
Transaction (транзакция)	Коллекция данных, относящихся к выполняемым операциям. Обеспечивает атомарные, последовательные, изолированные и надежные операции	Может быть связан с реестром и файловым вводом-выводом, чтобы сделать эти операции частью единой, более крупной операции
Enlistment (зачисление)	Связь между диспетчером ресурсов и транзакцией	Регистрируется вместе с транзакцией для получения уведомлений о ней. Зачисление может указать, какие уведомления должны быть сгенерированы
Resource Manager (RM) (диспетчер ресурсов)	Контейнер для транзакций и данных, с которыми они работают	Предоставляет интерфейс для клиентов для чтения и записи данных, как правило, в базе данных
Transaction Manager (TM) (диспетчер транзакций)	Контейнер для всех транзакций, являющихся частью соответствующих диспетчеров ресурсов. Являясь экземпляром журнала, знает обо всех состояниях транзакций, но не об их данных	Предоставляет инфраструктуру, через которую клиенты и диспетчеры ресурсов могут обмениваться данными, и предоставляет и координирует восстановительные операции после аварии. Клиенты используют TM для транзакций; Диспетчеры ресурсов (RM) используют TM для зачислений

ЭКСПЕРИМЕНТ: ВЫВОД СПИСКА ДИСПЕТЧЕРОВ ТРАНЗАКЦИЙ

Windows поставляется со встроенным инструментальным средством под названием Ktmutil.exe, которое позволяет вам просматривать происходящие в данное время транзакции, а также зарегистрированные в системе диспетчеры транзакций (и заставить сделать вывод о происходящих транзакциях). В этом эксперименте вы будете использовать данное средство для вывода диспетчеров транзакций, которые обычно можно увидеть на Windows-машине.

Начните с вызова командной строки с повышенными привилегиями и введите команду:

```
Ktmutil.exe tm list
```

Система Windows должна вывести примерно следующую информацию:

```
C:\Windows\system32>ktmutil tm list
TmGuid                                TmLogPath
------------------------------------ ----------------------------------------
```

продолжение ⇩

```
{fef0dc5f-0392-11de-979f-002219dd8c25} \Device\HarddiskVolume2\$Extend\$RmMetadata\
$TxfLog\$TxfLog::KtmLog
{fef0dc63-0392-11de-979f-002219dd8c25} \Device\HarddiskVolume1\$Extend\$RmMetadata\
$TxfLog\$TxfLog::KtmLog
{5e68e4aa-129e-11e0-8635-806e6f6e6963} \Device\HarddiskVolume2\Windows\
ServiceProfiles\NetworkService\ntuser.dat{5e68e4a8-129e-11e0-8635-806e6f6e6963}.TM
{5e68e4ae-129e-11e0-8635-005056c00008} \Device\HarddiskVolume2\Windows\
ServiceProfiles\LocalService\ntuser.dat{5e68e4ac-129e-11e0-8635-005056c00008}.TM
{51ce23c9-0d6c-11e0-8afb-806e6f6e6963} \SystemRoot\System32\Config\TxR\{51ce23c7-0d6c-
11e0-8afb-806e6f6e6963}.TM
{51ce23ee-0d6c-11e0-8afb-005056c00008} \Device\HarddiskVolume2\Users\markruss\
ntuser.dat{51ce23ec-0d6c-11e0-8afb-005056c00008}.TM
{51ce23f2-0d6c-11e0-8afb-005056c00008} \Device\HarddiskVolume2\Users\markruss\
AppData\Local\Microsoft\Windows\UsrClass.dat{51ce23f0-0d6c-11e0-8afb-005056c00008}.TM
```

Поддержка горячих исправлений

Перезагрузка машины для вступления в силу самых последних исправлений может означать существенные простои для сервера, именно поэтому Windows поддерживает метод внесения исправлений в ходе выполнения, который называется *горячим исправлением* (или просто hotpatch), в отличие от *холодного исправления*, которое требует перезагрузки. Горячее исправление не просто позволяет файлам быть переписанными в ходе выполнения, вместо этого оно включает сложную серию операций, которые могут быть запрошены (и объединены). Эти операции перечислены в табл. 3.28.

Таблица 3.28. Операции горячих исправлений

Операции	Значения	Использование
Переименование образа	Замена DLL-библиотеки, находящейся на диске и используемой в данный момент другими приложениями, или замена драйвера, находящегося на диске и в данный момент загруженного ядром	Когда в замене в пользовательском режиме нуждается целая библиотека, ядро может обнаружить, какие процессы и службы на нее ссылаются, выгрузить их, а затем обновить DLL и перезапустить программы и службы (что делается через *диспетчер перезапуска*). Когда нужно заменить драйвер, ядро может выгрузить драйвер (драйверу требуется процедура выгрузки), обновить его, а затем снова загрузить этот драйвер
Смена объекта	Проведение атомарного переименования объекта в пространстве имен каталога объектов	Когда файл (как правило, *известная DLL*) должен быть атомарно переименован, но не повлияв на любой процесс, который может его использовать (таким образом, чтобы процесс мог тут же приступить к использованию нового файла, используя старый дескриптор и не требуя перезапуска приложения)

Операции	Значения	Использование
Исправление функции	Замена кода одной или нескольких функций внутри файла образа другой версией	Если DLL или драйвер не может быть заменен или переименован в ходе выполнения программы, то функции в образе могут быть исправлены напрямую. При каждом вызове старой функции осуществляется безусловный переход к подвергаемой горячему исправлению DLL-библиотеке, содержащей новый код
Обновление системной DLL-библиотеки	Перезагрузка объекта раздела с отображенной памятью для Ntdll.dll	Относящаяся к системе библиотека Ntdll.dll загружается только один раз в процессе загрузки системы, а затем просто дублируется в адресное пространство каждого нового процесса. Если она подверглась горячему исправлению, система должна обновить этот раздел, чтобы загрузить новую версию

Хотя горячие исправления используют внутренние механизмы ядра, их фактическая реализация не отличается от реализации холодных исправлений. Исправление доставляется через систему обновлений Windows — Windows Update, как правило, в виде исполняемого файла, содержащего программу под названием Update.exe, которая извлекает исправление и запускает процесс обновления. Но для горячих исправлений будет еще и дополнительный файл горячего исправления с расширением .hp. Этот файл содержит специальный PE-заголовок, который называется .HOT1. В этом заголовке содержится структура данных, в которой дается описание различных дескрипторов исправлений, имеющихся внутри файла. Каждый из этих дескрипторов идентифицирует смещение в исходном файле, по которому следует внести исправление, механизм проверки (например, простое сравнение со старыми данными, контрольная сумма или хэш) и новые данные для внесения исправления. Ядро проводит разбор дескрипторов и применяет соответствующие модификации. В случае использования защищенного процесса (см. главу 5) и других образов, имеющих цифровую подпись, горячее исправление должно также иметь цифровую подпись, чтобы предотвратить применения ложных исправлений к важным файлам или процессам.

Горячее исправление времени компиляции поддерживает работу за счет добавления 7 дополнительных байт в начале каждой функции — 4 считаются частью окончания предыдущей функции, а 2 — частью пролога функции, то есть ее начала. Вот пример функции, которая была создана с информацией о горячем исправлении:

```
lkd> u nt!NtCreateFile - 5
nt!FsRtlTeardownPerFileContexts+0x169:
82227ea5 90              nop
82227ea6 90              nop
82227ea7 90              nop
82227ea8 90              nop
82227ea9 90              nop
nt!NtCreateFile:
82227eaa 8bff            mov     edi,edi
```

ПРИМЕЧАНИЕ

Поскольку файл горячего исправления также включает исходные данные, механизм применения горячего исправления может быть также использован для отмены установки горячего исправления без остановки машины.

Обратите внимание на то, что пять инструкций nop ничего не делают, а инструкция mov edi, edi в начале функции NtCreateFile также по своей сути бессмысленна — при ее выполнении не проводится никакой операции, реально изменяющей состояние. Благодаря доступности 7 байтов пролог функции NtCreateFile может быть преобразован в короткий безусловный переход к буферу из пяти доступных инструкций, которые затем превращаются в инструкцию близкого перехода в исправляемой процедуре. Вот как выглядит функция NtCreateFile после того, как она подверглась горячему исправлению:

```
lkd> u nt!NtCreateFile - 5
nt!FsRtlTeardownPerFileContexts+0x169:
82227ea5 e93d020010      jmp     nt_patch!NtCreateFile (922280e7)
nt!NtCreateFile:
82227eaa ebfc            jmp     nt!FsRtlTeardownPerFileContexts+0x169 (82227ea5)
```

Этот метод позволяет использовать только два дополнительных байта для каждой функции за счет безусловного перехода в заполнение для выравнивания предыдущей функции, которое у нее, скорее всего, все равно будет в ее конце.

У функциональных возможностей горячих исправлений есть ряд ограничений. К ним относятся:

- Исправления, которые могут блокироваться приложениями сторонних разработчиков, представляющих собой программы обеспечения безопасности или которые могут быть несовместимы с работой приложений сторонних разработчиков.
- Исправления, которые изменяют имеющуюся в файле таблицу экспорта или таблицу импорта.
- Исправления, изменяющие структуру данных, устраняющие бесконечные циклы или имеющие встроенный ассемблерный код.

Защита ядра от исправлений

Некоторые 32-разрядные драйверы устройств изменяют поведение Windows неподдерживаемыми способами. Например, они исправляют таблицу системных вызовов для перехвата этих вызовов или исправляют образ ядра в памяти для добавления функциональных возможностей определенным внутренним функциям. Вскоре после выпуска 64-разрядной Windows для x64 и перед тем, как была разработана богатая экосистема сторонних производителей, Microsoft увидела благоприятный момент для сохранения стабильности 64-разрядной Windows. Для предотвращения подобных изменений в x64 Windows реализована защита ядра от исправлений — Kernel Patch Protection (KPP), известная также как PatchGuard. Задача KPP в системе аналогична смыслу ее названия — она пытается не допустить применения обычных технологий исправления системы или внесения в нее каких-либо программных изменений. В табл. 3.29 перечислены те компоненты или структуры, которые находятся под защитой, и цели этой защиты.

Таблица 3.29. Компоненты, защищенные с помощью KPP

Компонент	Допустимое использование	Потенциальное вредоносное использование
Ntoskrnl.exe, Hal.dll, Ci.dll, Kdcom.dll, Pshed.dll, Clfs.sys, Ndis.sys, Tcpip.sys	Kernel, HAL и их зависимости. Самый нижний уровень сетевого стека	Исправление кода в ядре и (или) в HAL с целью подрыва нормальной работы и поведения. Внесение исправлений в Ndis.sys с целью незаметного добавления лазеек в открытых портах
Global Descriptor Table (GDT) (глобальная таблица дескрипторов)	Аппаратная защита центрального процессора для реализации кольцевых уровней привилегий (Ring 0 против Ring 3)	Возможность установки шлюза вызова (callgate), механизма центрального процессора, с помощью которого пользовательский код (Ring 3) может проводить операции с привилегиями ядра (Ring 0)
Interrupt Descriptor Table (IDT) (таблица дескрипторов прерываний)	Таблица считывается центральным процессором для доставки векторов прерывания к правильной процедуре обработки	Вредоносные драйверы могут перехватывать файловые вводы-выводы напрямую на уровне прерываний или захватывать ошибки обращений к страницам для того, чтобы скрыть содержимое памяти. Руткиты могут перехватывать обработчик INT2E для захвата всех системных вызовов из одной точки
System Service Descriptor Table (SSDT) (таблица дескрипторов системных служб)	Таблица, содержащая массив указателей для каждого обработчика системных вызовов	Руткиты могут изменять вывод или ввод вызовов из пользовательского режима и скрывать процессы, файлы или разделы реестра
Processor Machine State Registers (MSRs) (процессорные регистры состояния машины)	LSTAR MSR используется для установки обработчика инструкции SYSENTER и (или) инструкции SYSCALL, используемых для системных вызовов	LSTAR может быть переписан вредоносным драйвером для предоставления единого захвата для всех системных вызовов, выполняемых в системе
Указатели на функции KdpStub, KiDebugRoutine, KdpTrap	Используется для настроек в процессе выполнения тех мест, куда должны доставляться исключения на основе того, подключен ли удаленно к машине отладчик ядра	Значение указателей может быть переписано вредоносным руткитом для получения контроля над системой в предопределенные моменты времени и выполнения незаметных фоновых задач

продолжение ⇩

Таблица 3.29 *(продолжение)*

Компонент	Допустимое использование	Потенциальное вредоносное использование
PsInvertedFunctionTable	Кэширует каталоги исключений, используемые на x64, позволяя осуществлять быстрое отображение между кодом, в котором произошло исключение, и его обработчиком	Может быть использовано для получения контроля над системой в ходе обработки исключений несвязанного системного кода, включая собственный код исключений KPP, отвечающий в первую очередь за выявление изменений
Стеки ядра	Хранят аргументы функции, стек вызовов (куда функция должна вернуть управление) и переменные	Драйвер может выделить память на стороне, настроить ее в качестве стека ядра для потока, а затем манипулировать ее содержимым для перенаправления вызовов и параметров
Типы объектов	Определения разнообразных объектов (например, процессов и файлов), поддерживаемых системой с помощью диспетчера объектов	Может использоваться как часть технологии под названием DKOM (Direct Kernel Object Modification — непосредственное изменение объекта ядра) для изменения поведения системы, например, путем захвата функций обратного вызова объекта, которые зарегистрированы каждым типом объектов
Другие	Код, относящийся к проверке наличия ошибок в системе в случае нарушения режима KPP, выполняющий DPC-вызовы, запускающий таймеры, связанные с KPP и т. д.	Путем изменения определенных частей системы, используемых KPP, вредоносные драйверы могут предпринимать попытки замалчивания, игнорирования или применения какого-нибудь другого способа нарушения работы KPP

ПРИМЕЧАНИЕ

Поскольку на определенных 64-разрядных процессорах Intel реализован немного другой набор функций архитектуры x64, ядру для обхода недостатков инструкции упреждающей выборки (prefetch) необходимо в процессе выполнения кода осуществлять исправление этого кода. KPP может не допустить исправление ядра даже на таких процессорах, путем исключения этих специфических исправлений из числа обнаруживаемых. Кроме того, из-за наличия в гипервизоре (Hyper-V) так называемых просвещений (дополнительная информация о гипервизоре уже давалась ранее в этой главе), определенные функции в ядре, например процедура смены контекста, исправляются в процессе загрузки системы. Эти исправления также допускаются с весьма явной проверкой, чтобы убедиться в том, что они являются известными исправлениями для версий, просвещенных об использовании гипервизора.

Когда KPP обнаруживает изменение в любой из упомянутых структур (а также при проведении проверок целостности некоторых других внутренних структур), она вводит систему в аварийное состояние с кодом 0x109 —CRITICAL_STRUCTURE_CORRUPTION (критическое изменение структуры).

Сторонние разработчики, использующие технологии, сдерживаемые KPP, могут воспользоваться следующими поддерживаемыми технологиями:

- Мини-фильтрами файловой системы для захвата всех файловых операций, включая загрузку файлов образов и DLL-библиотек, которые могут быть перехвачены для очистки от вредоносного кода на лету или для блокирования чтения известных вредоносных исполняемых файлов.
- Уведомлениями фильтра реестра (см. главу 4) для захвата всех операций с реестром. Программы обеспечения безопасности могут заблокировать изменения критических частей реестра, а также эвристически определить вредоносное программное обеспечение за счет шаблонов обращения к реестру или известных «плохих» разделов реестра.
- Уведомлениями процесса (см. главу 5). Программы обеспечения безопасности могут отслеживать выполнение и завершение всех процессов и потоков системы, а также загружаемые и выгружаемые DLL-библиотеки. Используя расширенные уведомления, добавленные для поставщиков антивирусов и других программ обеспечения безопасности, они также могут блокировать запуск процесса.
- Фильтрацией диспетчера объектов (см. раздел, посвященный диспетчеру объектов). Программы обеспечения безопасности могут удалять некоторые права доступа, предоставленные процессам и (или) потокам для защиты своих собственных утилит от определенных операций.

После включения KPP выключить ее уже невозможно. Поскольку разработчики драйверов устройств могут в ходе отладки нуждаться во внесении изменений в запущенную систему, KPP не включается, если система загружается в режиме отладки с действующим подключением отладки ядра.

Целостность кода

Целостность кода — это механизм Windows, проводящий аутентификацию целостности и источника исполняемых образов (таких как приложения, DLL-библиотеки или драйверы) путем проверки достоверности цифрового сертификата, содержащегося внутри ресурсов образов. Этот механизм работает в единой связке с политиками системы, определяя порядок принудительного применения цифровых подписей. Одной из таких политик является политика подписей кода режима ядра — Kernel Mode Code Signing (KMCS), — требующая, чтобы код режима ядра был подписан действующим сертификатом аутентификации — Authenticode certificate, — внедренным одной из нескольких признанных организаций, подписывающих код, например Verisign или Thawte.

Для решения вопросов обратной совместимости политика KMCS полностью соблюдается только на 64-разрядных машинах, поскольку эти драйверы для запуска на данной архитектуре Windows должны быть в недавнем времени перекомпилированы. Это, в свою очередь, означает, что компания или частное лицо

по-прежнему отвечает за поддержку драйвера и может его подписать. Но на 32-разрядных машинах многие старые устройства поставляются с устаревшими драйверами, возможно, от прекративших свою деятельность компаний, поэтому поставить подпись в таких драйверах иногда не представится возможности. На рис. 3.43 показано предупреждение, выводимое на машине с 64-разрядной Windows, на которую пытаются установить драйвер без подписи.

ПРИМЕЧАНИЕ

В Windows также имеется еще одна политика подписи драйверов, являющаяся частью диспетчера устройств Plug and Play. Эта политика применяется только к драйверам устройств Plug and Play, и в отличие от политики подписей кода режима ядра, она может быть настроена на разрешение использования драйверов устройств Plug and Play, не имеющих подписи (но не на 64-разрядных системах, где приоритетом пользуется политика KMCS).

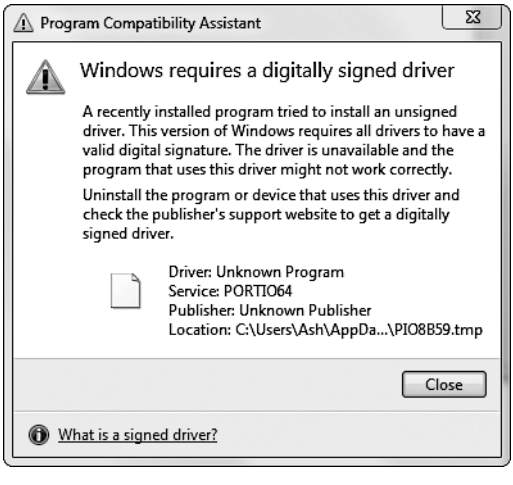


Рис. 3.43. Предупреждение, выводимое при попытке установить 64-разрядный драйвер, не имеющий подписи

Даже на 32-разрядной версии Windows система проверки целостности при загрузке неподписанного драйвера заносит это событие в журнал событий целостности кода (Code Integrity event log).

ПРИМЕЧАНИЕ

Приложения, использующие защищенный медиа-путь, могут также запрашивать у ядра состояние его целостности, куда включается информация, были или не были загружены в систему 32-разрядные драйверы без подписи. В подобных сценариях они позволяют отключить проигрывание защищенного медиа-контента высокого разрешения в качестве метода гарантии безопасности и надежности зашифрованных потоков.

Но механизм целостности кода не останавливает свою работу после загрузки драйвера. Существуют также меры по аутентификации содержимого каждой

страницы образа для исполняемых страниц. Это требует использования специального флага при подписи двоичного исполняемого кода драйвера и приводит к созданию каталога с криптографическим хэшем для каждой исполняемой страницы, на которой будет находиться драйвер. (На центральном процессоре страницы являются единицей защиты.) Этот метод допускает определение изменения существующего драйвера, которое могло произойти либо во время выполнения кода за счет работы другого драйвера или через атаку, предпринятую в отношении страничного файла или файла, созданного в процессе гибернации (содержимое памяти редактируется на диске и затем снова загружается в память). Генерирование таких хэшей для каждой страницы является требованием для новой модели фильтрации, а также для компонентов защищенного медиапути.

Заключение

В данной главе были рассмотрены самые основные системные механизмы, на которых построена исполняющая система Windows. В следующей главе будут рассмотрены три важных механизма, связанных с управлением инфраструктурой Windows: реестр, службы и инструментарий управления Windows — Windows Management Instrumentation (WMI).

Глава 4. Механизмы управления

В этой главе рассматриваются основные механизмы операционной системы Microsoft Windows, играющие основную роль в ее управлении и конфигурации:

- Реестр.
- Службы.
- Единый диспетчер фоновых процессов (Unified Background Process Manager).
- Инструментальные средства управления Windows (Windows Management Instrumentation).
- Инфраструктура диагностики Windows (Windows Diagnostics Infrastructure).

Реестр

Реестр играет ключевую роль в конфигурации систем Windows и в управлении ими. Он является хранилищем как для общесистемных настроек, так и для настроек для каждого пользователя. Хотя многие считают реестр некими статичными данными, хранящимися на жестком диске, как будет показано в данном разделе, реестр также является окном в различные находящиеся в памяти структуры, обслуживаемые исполняющей системой и ядром Windows.

Начнем с обзора структуры реестра, рассмотрения поддерживаемых им типов данных и краткого изучения ключевой информации, которую Windows содержит в реестре. Затем заглянем во внутренности диспетчера конфигурации — компонента исполняющей системы, ответственного за реализацию базы данных реестра. Мы также рассмотрим внутреннюю дисковую структуру реестра, порядок извлечения конфигурационной информации по запросам приложений и меры защиты этой важной системной базы данных.

Просмотр и изменение реестра

Скорее всего, вам никогда не придется редактировать реестр напрямую: хранящиеся в реестре настройки приложений и систем, которые могут потребовать производимых вручную изменений, должны иметь соответствующий пользовательский интерфейс для их изменения. Но, как вы уже неоднократно видели в данной книге, у некоторых дополнительных и отладочных настроек нет редактирующего пользовательского интерфейса. Поэтому в состав Windows включены инструменты как имеющие графический пользовательский интерфейс — graphical user interface (GUI), — так и работающие из командной строки, позволяющие просматривать содержимое реестра и вносить в него изменения.

Windows поставляется с одним основным GUI-инструментом для редактирования реестра — Regedit.exe — и с несколькими инструментами для работы из командной строки. Например, такая программа, как Reg.exe, позволяет импортировать и экспортировать разделы, сохранять их копию и восстанавливать их из копии, а также проводить их сравнение, вносить в них правки и удалять разделы и параметры. Она также может устанавливать флаги, использующиеся

в UAC-виртуализации, или запрашивать их состояние. А такая программа, как Regini.exe, позволяет импортировать данные реестра на основе текстовых файлов, содержащих данные конфигурации в формате ASCII или Unicode.

Комплект для разработки драйверов — Windows Driver Kit (WDK) — также предоставляет свободно распространяемый компонент Offreg.dll, в котором содержится библиотека Offline Registry Library. Эта библиотека позволяет загружать файлы кустов реестра в их двоичном формате и проводить операции над самими файлами, пропуская обычную логическую загрузку и отображение, которые требуются Windows для операций с реестром. Главным образом она применяется для содействия автономному доступу к реестру, например, в целях проверки целостности и приемлемости данных. Она также может предоставить преимущества в производительности, если базовые данные не предназначены для просмотра в системе, поскольку доступ к ним осуществляется через локальный файловый ввод-вывод, а не через системные вызовы реестра.

Использование реестра

Данные конфигурации считываются в следующих четырех основных случаях:

- В ходе процесса начальной загрузки начальный загрузчик считывает данные конфигурации и список загружаемых драйверов устройств для загрузки в память до инициализации ядра. Поскольку загрузочная конфигурационная база данных — Boot Configuration Database (BCD) — на самом деле хранится в кусте реестра, могут последовать возражения, что обращение к реестру происходит даже еще раньше, когда диспетчер загрузки (Boot Manager) выводит список операционных систем.
- В ходе процесса загрузки ядра это ядро считывает настройки, определяющие, какие драйверы устройств нужно загружать и как сами по себе сконфигурированы различные элементы системы, например диспетчер памяти и диспетчер процессов, настраивая при этом поведение системы.
- В процессе входа в систему Explorer и другие компоненты Windows считывают предпочтения того или иного пользователя, включая названия сетевых дисков, обои рабочего стола, экранные заставки, поведение меню, размещение значков, и что, наверное, более существенно, какие программы запустить и к каким файлам было обращение в последнее время.
- В процессе своего запуска приложения считывают общесистемные настройки, например список дополнительно установленных компонентов и данных о лицензировании, а также индивидуальные настройки пользователя, которые могут включать размещение меню и панели инструментов, а также список тех документов, к которым он недавно обращался.

Разумеется, реестр может также подвергаться чтению и в других случаях, например в ответ на изменение параметра или раздела реестра. Хотя реестром предоставляются асинхронные функции обратного вызова, являющиеся предпочтительным способом уведомления об изменениях, некоторые приложения производят постоянное отслеживание настроек своей конфигурации в реестре посредством опросов и автоматически принимают в расчет измененные настройки. Но, в общем, на простаивающей системе не должно быть никакой активности, свя-

занной с реестром, поэтому такие приложения идут вразрез со сложившейся практикой. Отличным средством отслеживания такой активности и приложения или приложений, нарушающих правила, является Process Monitor от Sysinternals.

Чаще всего изменения в реестр вносятся в следующих случаях:

- Хотя это и не считается изменением, но начальная структура реестра и многие установки по умолчанию определяются версией-прототипом реестра, поставляемой на носителе, использующимся для установки Windows, и копируемой в новую установку.
- Утилиты установки приложения создают настройки приложения по умолчанию и те настройки, которые отражают выбранные при установке настройки конфигурации.
- В ходе установки драйвера устройства система обслуживания устройств Plug and Play создает настройки реестра, которые сообщают диспетчеру ввода-вывода о том, как запустить драйвер, и создает другие настройки, задающие конфигурацию операций драйвера
- При изменении настроек приложения или системы через пользовательские интерфейсы эти изменения, чаще всего, сохраняются в реестре.

Типы данных реестра

Реестр является базой данных со структурой, схожей со структурой дискового тома. Реестр содержит разделы, похожие на каталоги диска, и параметры, которые можно сравнить с файлами на диске. Раздел является контейнером, который состоит из других разделов (подразделов) или из параметров, в параметрах хранятся данные. Самые верхние разделы считаются корневыми (рассматривая данную тему, мы будем считать, что слова «подраздел» и «раздел» взаимозаменяемы).

Как разделы, так и параметры позаимствовали соглашение о своих именах у файловой системы. Следовательно, уникально идентифицировать параметр mark, который хранится в разделе trade, можно с помощью записи trade\mark. Единственным исключением в этой схеме имен является безымянный параметр каждого раздела. Редактор реестра Regedit показывает безымянные параметры с пометкой (По умолчанию).

В параметрах хранятся различные типы данных, которые могут относиться к одному из 12 типов, перечисленных в табл. 4.1. Большинство значений параметров относятся к типам REG_DWORD, REG_BINARY или REG_SZ. Параметры типа REG_DWORD могут содержать числа или булевы значения (включен-выключен). Параметры типа REG_BINARY могут содержать числа, превышающие 32 разряда или необработанные данные, например зашифрованные пароли; Параметры типа REG_SZ содержат строки (разумеется, в формате Unicode), которые представляют такие элементы, как имена, имена файлов, пути и типы.

Тип REG_LINK представляет особый интерес, поскольку он позволяет разделу в явном виде указывать на другой раздел. Когда реестр проходится по ссылке, поиск пути продолжается по цели этой ссылки. Например, если у \Root1\Link есть параметр REG_LINK со значением \Root2\RegKey, а параметр RegKey содержит параметр RegValue, то RegValue идентифицируется двумя путями: \Root1\Link\RegValue и \Root2\RegKey\RegValue. Как объясняется в следующем раз-

деле этой главы, Windows нередко использует ссылки реестра: 3 из 6 корневых разделов реестра дают ссылки на подразделы внутри трех корневых разделов, на которые не имеется ссылок.

Таблица 4.1. Типы параметров реестра

Тип параметра	Описание
REG_NONE	Тип параметра отсутствует
REG_SZ	Строка в формате Unicode фиксированной длины
REG_EXPAND_SZ	Строка в формате Unicode переменной длины, у которой могут быть встроенные переменные среды
REG_BINARY	Двоичные данные произвольной длины
REG_DWORD	32-разрядное число
REG_DWORD_BIG_ENDIAN	32-разрядное число, в котором сначала следует старший байт
REG_LINK	Символическая ссылка в формате Unicode
REG_MULTI_SZ	Массив из строк в формате Unicode, завершающихся нулевым байтом
REG_RESOURCE_LIST	Описание ресурса оборудования
REG_FULL_RESOURCE_DESCRIPTOR	Описание ресурса оборудования
REG_RESOURCE_REQUIREMENTS_LIST	Потребности ресурса
REG_QWORD	64-разрядное число

Логическая структура реестра

Схему организации реестра можно составить на основе хранящихся в нем данных. В реестре имеется 6 корневых разделов (добавить к ним новые корневые разделы или удалить уже существующие невозможно), в которых хранится информация, показанная в табл. 4.2.

Почему имена корневых разделов начинаются с буквы H? Потому что имена корневых объектов представляют собой дескрипторы Windows — handles (H), относящиеся к разделам — keys (KEY). Как уже упоминалось в главе 1 «Общие представления и инструментальные средства», HKLM является аббревиатурой для HKEY_LOCAL_MACHINE. В табл. 4.3 перечислены все корневые разделы и их аббревиатуры. Содержимое и цели каждого из шести корневых разделов подробно рассматриваются в следующих разделах.

Таблица 4.2. Шесть корневых разделов

Корневой раздел	Описание
HKEY_CURRENT_USER	Хранит данные, связанные с текущим пользователем, вошедшим в систему
HKEY_USERS	Хранит информацию обо всех учетных записях, имеющихся на машине

продолжение ⇩

Таблица 4.2 *(продолжение)*

Корневой раздел	Описание
HKEY_CLASSES_ROOT	Хранит файловые связи и информацию о регистрации объектов, относящихся к модели компонентных объектов — Component Object Model (COM)
HKEY_LOCAL_MACHINE	Хранит информацию, связанную с системой
HKEY_PERFORMANCE_DATA	Хранит информацию о производительности
HKEY_CURRENT_CONFIG	Хранит определенную информацию о текущем профиле оборудования

Таблица 4.3. Корневые разделы реестра

Корневой раздел	Аббревиатура	Описание	Ссылка
HKEY_CURRENT_USER	HKCU	Указывает на профиль текущего, вошедшего в систему пользователя	Подраздел в разделе 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 файловые связи и информацию о регистрации COM-объектов	Не является прямой ссылкой, а представляет собой объединенное представление о разделе HKLM\SOFTWARE\Classes и о разделе HKEY_USERS\<*SID*>\SOFTWARE\Classes
HKEY_LOCAL_MACHINE	HKLM	Содержит глобальные настройки для машины	Не является ссылкой
HKEY_CURRENT_CONFIG	HKCC	Содержит текущий профиль оборудования	HKLM\SYSTEM\CurrentControlSet\Hardware Profiles\Current
HKEY_PERFORMANCE_DATA	HKPD	Содержит показания счетчиков производительности	Не является ссылкой

HKEY_CURRENT_USER

Корневой раздел HKCU содержит данные, относящиеся к персональным настройкам и программной конфигурации локально вошедшего в систему пользователя. Он указывает на пользовательский профиль текущего вошедшего в систему пользователя, находящийся на жестком диске в файле \Users\<*имя_пользователя*>\Ntuser.dat (см. раздел «Внутреннее устройство реестра»). При загрузке профиля пользователя (например, во время входа в систему или когда служебный процесс запускается в контексте конкретного имени пользователя) создается раздел HKCU для отображения на раздел пользователя в разделе HKEY_USERS. В табл. 4.4 перечислены некоторые подразделы раздела HKCU.

Таблица 4.4. Подразделы HKEY_CURRENT_USER

Подраздел	Описание
AppEvents	Связи между событиями и звуками
Console	Настройки окна командной строки (например, ширина, высота и цветовые решения)
Control Panel	Заставка экрана, схема рабочего стола, настройки клавиатуры и мыши, а также доступность и региональные настройки
Environment	Определения переменных среды окружения
EUDC	Информация о символах, определенных конечным пользователем
Identities	Информация об учетной записи почты Windows
Keyboard Layout	Настройки раскладки клавиатуры (например, US. или UK.)
Network	Настройки и отображения сетевого драйвера
Printers	Настройки подключения принтера
Software	Предпочтения пользователя в отношении программного обеспечения
Volatile Environment	Временные определения переменных среды окружения

HKEY_USERS

Раздел HKU содержит подраздел для каждого загруженного профиля пользователя и использует базу данных классов, зарегистрированных в системе. Он также содержит подраздел HKU\.DEFAULT, связанный с профилем системы (используется процессами, запущенными под учетной записью локальной системы, см. далее в разделе «Службы»). Этот профиль используется Winlogon, например, так, чтобы изменения в настройках фона рабочего стола, установленные в этом профиле, применялись к экрану входа в систему. Когда пользователь входит в систему в первый раз и его учетная запись не зависит от перемещаемого профиля домена (пользовательский профиль берется из центрального сетевого размещения в направлении контроллера домена), система создает профиль для его учетной записи, основа для которого берется у профиля, сохраненного в каталоге %SystemDrive%\Users\Default.

Место, в котором система хранит профили, определяется параметром реестра HKLM\Software\Microsoft\Windows NT\CurrentVersion\ProfileList\ ProfilesDirectory, значение которого по умолчанию установлено в %SystemDrive%\ Users. В разделе ProfileList также хранится список профилей, имеющихся в системе. Информация для каждого профиля размещена в подразделе, имя которого отображает идентификатор безопасности — security identifier (SID), — той учетной записи, которой соответствует профиль (см. главу 6). Данные, хранящиеся в разделе профиля, включают время последней загрузки профиля (которое хранится в параметре ProfileLoadTimeLow), двоичное представление SID учетной записи в параметре Sid и путь к кусту профиля на диске в каталоге ProfileImagePath. Windows выводит список профилей, сохраненных в системе, в диалоговом окне Профили пользователей (User Profiles), доступ к которому можно получить, щелкнув на пункте Параметры (Settings) в разделе Профили пользователей (User Profiles) вкладки Дополнительно (Advanced) в подразделе Дополнительные параметры системы (Advanced System Settings) раз-

дела Система (System) Панели управления (Control Panel). Диалоговое окно Профили пользователей (User Profiles) показано на рис. 4.1.

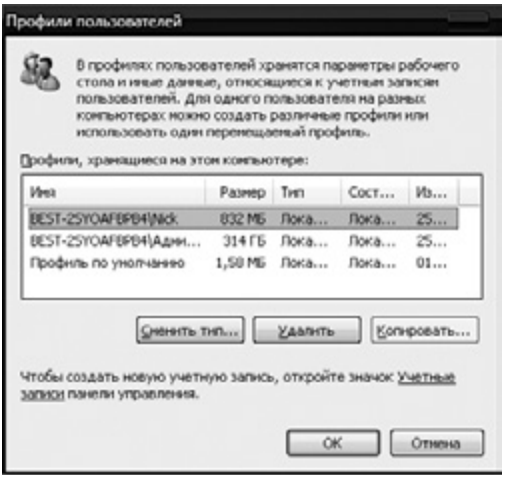


Рис. 4.1. Диалоговое окно управления Профили пользователей

ЭКСПЕРИМЕНТ: НАБЛЮДЕНИЕ ЗА ЗАГРУЗКОЙ И ВЫГРУЗКОЙ ПРОФИЛЕЙ

Используя команду Runas (запустить от имени) для запуска процесса в учетной записи, не являющейся текущей загруженной на машине учетной записью, можно увидеть, как профиль загружается в реестр, а затем выгружается из него. Как только будет запущен новый процесс, запустите программу Regedit и возьмите на заметку загруженный раздел профиля, имеющийся в разделе HKEY_USERS. После завершения процесса обновите данные в программе Regedit, нажав клавишу F5, и убедитесь в том, что этого профиля уже быть не должно. ■

HKEY_CLASSES_ROOT

Раздел HKCR состоит из трех типов информации:

- ассоциации с расширениями файлов;
- регистрации COM-классов;
- виртуализированный корневой раздел реестра системы для управления учетными записями пользователей — User Account Control (UAC) (см. главу 6).

Для каждого зарегистрированного расширения имени файла есть свой раздел. Большинство разделов содержат параметр типа REG_SZ, который указывает на другой раздел в HKCR, содержащий информацию, связанную с тем классом файлов, который представляет данное расширение.

Например, HKCR\.xls будет указывать на информацию о файлах Microsoft Office Excel в таком разделе, как HKCU\.xls\Excel.Sheet.8. В других разделах содержатся подробности конфигурации для COM-объектов, зарегистрированных в системе. Виртуализированный реестр UAC находится в разделе VirtualStore, который не имеет никакого отношения к другим разновидностям данных, хранящихся в HKCR.

Данные, хранящиеся в разделе HKEY_CLASSES_ROOT, поступают из двух источников:

- из данных о регистрации классов для отдельного пользователя, находящихся в разделе HKCU\SOFTWARE\Classes (который отображается на файл на жестком диске \Users\<*имя_пользователя*>\AppData\Local\Microsoft\Windows\Usrclass.dat);
- из данных о регистрации классов в масштабе всей системы, находящихся в разделе HKLM\SOFTWARE\Classes.

Причина отделения регистрационных данных, относящихся к каждому пользователю, от регистрационных данных, распространяемых на всю систему, объясняется тем, что эти настройки могут содержаться в перемещаемых профилях. Тем самым также закрывается дыра в системе безопасности: непривилегированный пользователь не может изменить или удалить разделы в общесистемной версии HKEY_CLASSES_ROOT и не может повлиять на функционирование приложений в системе. Непривилегированные пользователи и приложения могут считывать общесистемные данные и могут добавлять новые разделы и параметры (которые зеркально отображаются в их данных, предназначенных для отдельного пользователя), но изменять существующие разделы и параметры они могут только в своих частных данных.

HKEY_LOCAL_MACHINE

HKLM является корневым разделом, в котором содержатся общесистемные подразделы конфигурации: BCD00000000, COMPONENTS (загружаемый в динамическом режиме по мере необходимости), HARDWARE, SAM, SECURITY, SOFTWARE и SYSTEM.

Подраздел HKLM\BCD00000000 содержит информацию из базы данных загрузочной конфигурации — Boot Configuration Database (BCD), — загружаемую в качестве куста реестра. Эта база данных заменила файл Boot.ini, который использовался до выхода Windows Vista, и придала больше гибкости и изолированности в данные загрузочной конфигурации для отдельно взятой установки.

Каждая запись в BCD, например, об установке Windows или о настройках командной строки для установки, хранится в подразделе Objects: либо в виде объекта, на который ссылается GUID (в случае загрузочной записи), либо в виде числового подраздела, называемого элементом. Большинство таких простых элементов фигурирует в справочнике по BCD в библиотеке MSDN. Эти элементы определяют различные настройки командной строки или параметры загрузки. Параметр, связанный с каждым подразделом элемента, соответствует параметру для соответствующего ему ключа командной строки или параметра загрузки.

Утилита командной строки BCDEdit позволяет изменять BCD, используя символические имена для элементов и объектов. Она также предоставляет исчерпывающую справочную информацию по доступным настройкам загрузки, но, к сожалению, эта утилита работает только в локальном режиме. Поскольку реестр может быть открыт удаленно, а также импортирован из файла куста, вы можете модифицировать или прочитать BCD удаленного компьютера с помощью редактора реестра. В следующем эксперименте показано, как с помощью редактора реестра включить отладку ядра.

Подраздел HKLM\COMPONENTS содержит информацию, относящуюся к стеку обслуживания на основе компонентов — Component Based Servicing (CBS). Этот стек содержит различные файлы и ресурсы, являющиеся частью образа установки Windows — Windows installation image (используется пакетом автоматической установки — Automated Installation Kit — или предустановочным OEM-пакетом — OEM Preinstallation Kit) или активной установки. API-функции CBS, предназначенные для обслуживания, используют информацию, находящуюся в этом разделе, для идентификации установленных компонентов и получения информации относительно их конфигурации. Эта информация используется, когда компоненты устанавливаются, обновляются или удаляются либо в индивидуальном порядке (так называемыми модулями), либо группами (так называемыми пакетами). Для оптимизации системных ресурсов он загружается и выгружается только в динамическом режиме по мере надобности (этот раздел может стать слишком большим), если запрошено обслуживание стека CBS.

Подраздел HKLM\HARDWARE хранит в себе описания унаследованного системой оборудования и некоторого отображения аппаратных устройств на драйверы. На современных системах в нем, скорее всего, будут найдены только некоторые периферийные устройства, такие как клавиатура, мышь и данные ACPI BIOS. Инструментальное средство Диспетчер устройств (запускается щелчком на пункте Диспетчер устройств (Device Manager) в окне Система (System) Панели управления (Control Panel)) позволяет просмотреть информацию об оборудовании, зарегистрированную в реестре, получаемую простым чтением параметров в разделе HARDWARE (дерево HKLM\SYSTEM\CurrentControlSet\Enum).

В подразделе HKLM\SAM хранится информация о локальных учетных записях и группах, например пароли, определения групп и связи доменов. Системы Windows Server, работающие в качестве контроллеров доменов, хранят доменные учетные записи и группы в Active Directory, базе данных, в которой хранятся общедоменные настройки и сведения[1]. По умолчанию дескриптор безопасности в разделе SAM сконфигурирован таким образом, что к нему нет доступа даже с использованием учетной записи администратора.

В подразделе HKLM\SECURITY хранится общесистемная политика безопасности и назначения прав пользователям. Подраздел HKLM\SAM связан с подразделом SECURITY в подразделе HKLM\SECURITY\SAM. По умолчанию вы не можете просматривать содержимое HKLM\SECURITY или HKLM\SAM\SAM, поскольку настройки безопасности этих разделов разрешают доступ только с учетной записью System. Чтобы разрешить доступ администраторам, можно изменить дескриптор безопасности или воспользоваться PsExec для запуска Regedit с учетной записью локальной системы, если нужно заглянуть вовнутрь. Но это беглое знакомство вам вряд ли что-либо даст, поскольку данные не фигурируют ни в каких документах, а пароли зашифрованы с помощью однонаправленного отображения — то есть вы не можете определить пароль на основе его зашифрованной формы.

В подразделе HKLM\SOFTWARE Windows хранит общесистемную конфигурационную информацию, не требующуюся для загрузки системы. Кроме этого общесистемные установки, такие как пути к файлам и каталогам приложений,

[1] Active Directory в данной книге не рассматривается.

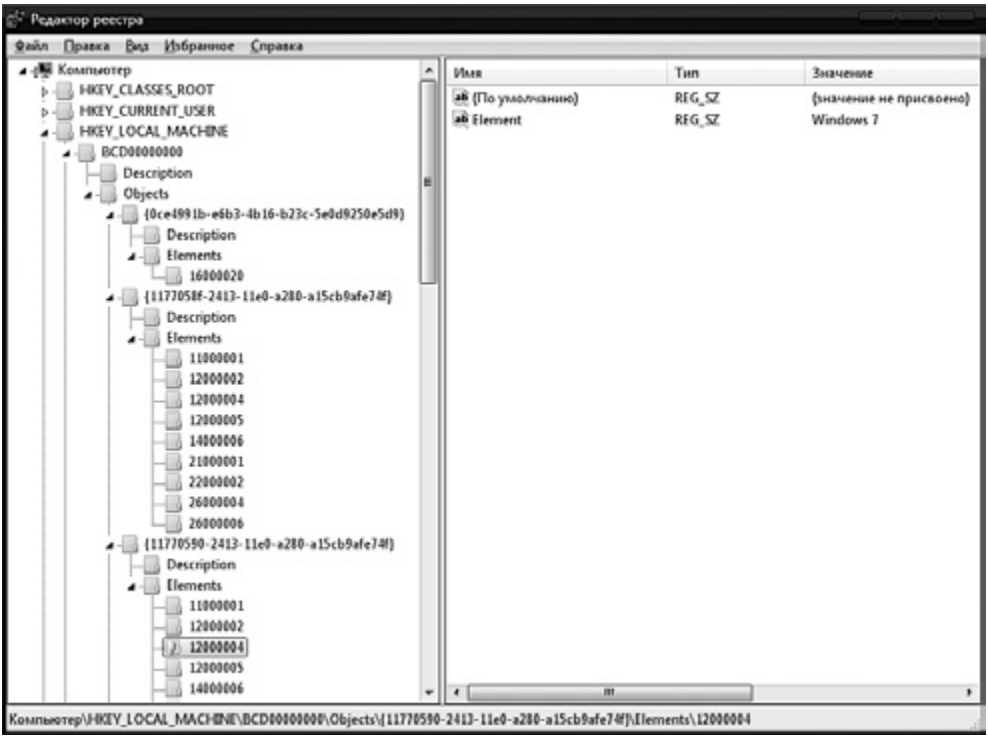
информацию о лицензировании и истечении срока использования здесь хранят и приложения сторонних разработчиков.

В подразделе HKLM\SYSTEM содержится общесистемная конфигурационная информация, необходимая для загрузки системы, например какие драйверы устройств нужно загружать и какие службы нужно запускать. Поскольку эта информация существенна для запуска системы, Windows также содержит в этом подразделе копию части этой информации под названием последней удачной конфигурации. Сохранение такой копии позволяет администратору выбрать предыдущий рабочий набор управления в том случае, когда конфигурационные изменения, внесенные в текущий набор управления, не дают системе загрузиться. Подробности того, когда Windows объявляет текущий набор управления «удачным», рассмотрены в следующем далее разделе «Признание загрузки и последняя удачная конфигурация».

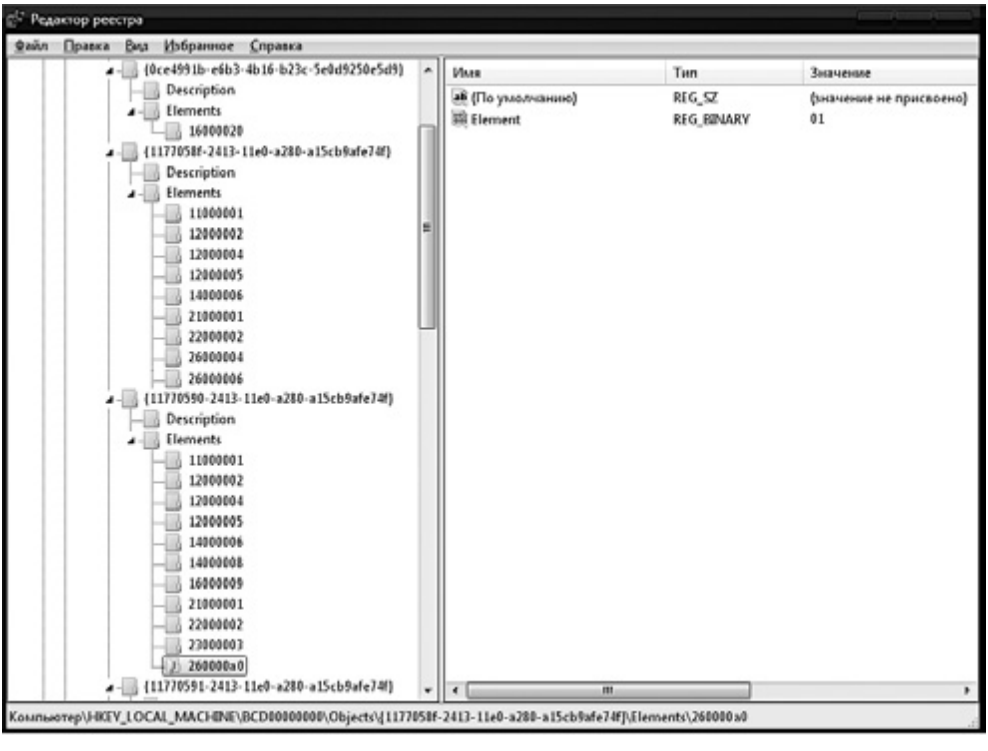
ЭКСПЕРИМЕНТ: АВТОНОМНОЕ ИЛИ УДАЛЕННОЕ РЕДАКТИРОВАНИЕ BCD

В данном эксперименте путем редактирования BCD внутри реестра будет включен режим отладки. Для достижения целей, поставленных в данном примере, вы будете редактировать локальную копию BCD, но особенностью этой технологии является то, что ее можно использовать в отношении куста BCD любой машины. Чтобы добавить ключ командной строки /DEBUG, выполните следующие действия:

1. Откройте редактор реестра, а затем перейдите к разделу HKLM\BCD00000000. Раскройте каждый подраздел, чтобы стали полностью видны числовые идентификаторы каждого раздела Elements.



2. Установите загрузочную запись для вашей установки Windows путем определения местоположения подраздела Description с параметром Type, имеющим значение 0x10200003, а затем проверьте идентификатор 0x12000004 в дереве Elements. В параметре Element этого подраздела нужно найти имя вашей версии Windows, например Windows 7. Если на вашей машине имеется более одной установки Windows, может понадобиться проверить элемент 0x22000002, который содержит путь, например \Windows.
3. Теперь, когда найден правильный GUID для вашей установки Windows, создайте новый подраздел в подразделе Elements для этого GUID и назовите его 0x260000a0. Если этот подраздел уже существует, то просто перейдите в него.
4. Если вам пришлось создать подраздел, то теперь создайте внутри него двоичный параметр под названием Element.
5. Отредактируете параметр, установив для него значение 01. Тем самым будет включен режим отладки ядра. Вот как должны выглядеть эти изменения.



ПРИМЕЧАНИЕ

Идентификатор 0x12000004 соответствует BcdLibraryString_ApplicationPath, а идентификатор 0x22000002 соответствует BcdOSLoaderString_SystemRoot. И наконец, добавленный вами идентификатор 0x260000a0 соответствует BcdOSLoaderBoolean_KernelDebuggerEnabled. Описание этих параметров дано в справочнике по BCD в библиотеке MSDN.

HKEY_CURRENT_CONFIG

Подраздел HKEY_CURRENT_CONFIG является всего лишь ссылкой на текущий профиль оборудования, сохраненный в разделе HKLM\SYSTEM\CurrentControlSet\Hardware Profiles\Current. Профили оборудования в Windows больше не поддерживаются, но раздел по-прежнему существует для поддержки устаревших приложений, работа которых может зависеть от их присутствия.

HKEY_PERFORMANCE_DATA

Реестр является механизмом, используемым в Windows для доступа к значениям счетчиков производительности, относящихся либо к операционной системе, либо к серверным приложениям. Одним из сопутствующих преимуществ предоставления доступа к счетчикам производительности через реестр является то, что удаленное отслеживание производительности работает «бесплатно», поскольку удаленный доступ к реестру легко обеспечивается с помощью обычных API-функций реестра.

Можно получить непосредственный доступ к информации счетчиков производительности в реестре, открыв специальный раздел HKEY_PERFORMANCE_DATA и запросив находящиеся в нем параметры. Просматривая реестр в редакторе реестра, вы этот раздел не найдете, он доступен только программным способом, через имеющиеся в Windows функции реестра, например через RegQueryValueEx. Информация о производительности на самом деле в реестре не хранится, функции реестра используют этот раздел для поиска информации у поставщиков данных о производительности.

Получить доступ к информации счетчиков производительности можно также путем использования функций помощника в получении данных о производительности — Performance Data Helper (PDH), доступных через API Performance Data Helper (Pdh.dll). На рис. 4.2 показаны компоненты, задействованные в получении доступа к информации счетчиков производительности.

Расширение для работы с реестром в режиме транзакций — Transactional Registry (TxR)

Благодаря диспетчеру транзакций ядра (см. главу 3) — Kernel Transaction Manager (KTM) — разработчики имеют доступ к простому API, который позволяет им реализовывать надежные средства восстановления после возникновения ошибок, которые могут быть связаны с такими не имеющими к реестру операциями, как операции работы с файлами или с базами данных.

Изменения реестра в режиме транзакций поддерживаются с помощью трех API-функций: RegCreateKeyTransacted, RegOpenKeyTransacted и RegDeleteKeyTransacted. Эти новые функции используют те же параметры, что и их аналоги, не использующие транзакций, если не считать добавления нового параметра дескриптора транзакции. Разработчик предоставляет этот дескриптор после вызова KTM-функции CreateTransaction.

После операции создания или открытия, проведенной в режиме транзакции, все последующие операции с реестром, например создание, удаление или изме-

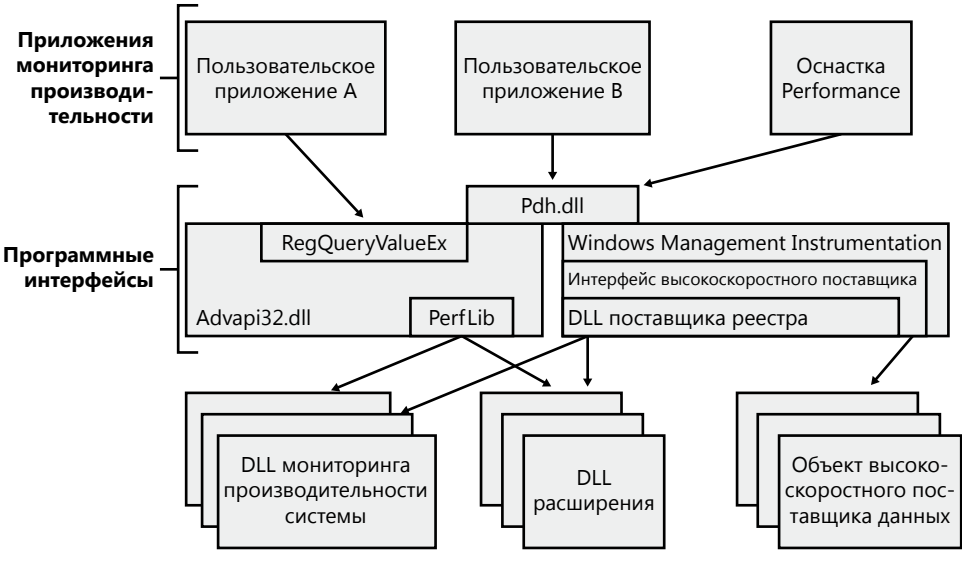


Рис. 4.2. Архитектура счетчиков производительности, обозначиваемых в реестре

нение параметров внутри раздела, будут также проводиться в режиме транзакции. Но операции над подразделами того раздела, работа над которым проходит в режиме транзакции, не будет автоматически проводиться в таком же режиме, именно поэтому существует третья API-функция — RegDeleteKeyTransacted. Она позволяет проводить удаление подразделов в режиме транзакции, чем обычно занимается функция RegDeleteKeyEx.

Данные для этих операций в режиме транзакции, подобно другим KTM-операциям, записываются в файлы регистрации с помощью служб общей файловой системы журналирования — common logging file system (CLFS). Пока сама транзакция не будет зафиксирована или отменена (как программно, так и в результате сбоев в электропитании или в системе), изменения разделов, параметров и другие изменения, связанные с реестром, выполненные с дескриптором транзакции, не будут видны внешним приложениям через API-функции, которые не имеют отношения к транзакциям. Кроме того, транзакции изолированы друг от друга, изменения, производимые внутри одной транзакции, не будут видны из других транзакций или за пределами транзакции, пока та не будет зафиксирована.

ПРИМЕЧАНИЕ

В случае конфликта программа записи, которая не использует транзакцию, отменит текущую транзакцию, например, если параметр был создан внутри транзакции или позже, когда транзакция еще сохраняла активность, а программа записи, не использующая транзакцию, пытается создать параметр в том же разделе. Проведение операции, не использующей транзакцию, завершится успехом, а все операции в конфликтной транзакции будут отменены.

Уровень изоляции («I» в ACID), реализуемый диспетчерами ресурсов TxR, относится к типу read-commit (позволяет читать только зафиксированные данные), это означает, что изменения становятся доступными другим читающим программам (использующим транзакцию или нет) сразу же после фиксации транзакции. Этот механизм важен для тех, кто знаком с транзакциями в базах данных, где уровень изоляции относится к типу предсказуемых чтений — predictable-reads (в литературе по базам данных — cursor-stability — чтение по установленному курсору). При использовании уровня изоляции predictable-reads, после того как будет прочитан параметр внутри транзакции, последующие чтения будут возвращать те же данные. Read-commit не дает такой гарантии. Одним из последствий является то, что реестровые транзакции не могут использоваться для операций «атомарного» инкремента-декремента в отношении параметра реестра.

Для постоянных изменений реестра приложение, использующее обработчик транзакции, должно вызывать KTM-функцию CommitTransaction. Если приложение решает отменить изменения, например, из-за сбоя в пути, оно может вызвать API-функцию RollbackTransaction. Затем изменения станут видны также и через обычные API-функции реестра.

ПРИМЕЧАНИЕ

Если дескриптор транзакции, созданный с помощью функции CreateTransaction, закрыт до фиксации транзакции (а других дескрипторов, открытых для этой транзакции, не существует), система проведет откат этой транзакции.

Кроме использования поддержки CLFS, предоставляемой KTM, TxR также сохраняет свои собственные внутренние файлы регистрации в папке %SystemRoot%\System32\Config\Txr на системном томе; у этих файлов имеется расширение .regtrans-ms, и по умолчанию они скрыты. Даже при отсутствии установленных приложений сторонних производителей на вашей системе, скорее всего, будут файлы в этом каталоге, потому что TxR используется Центром обновления Windows — Windows Update and Component Based Servicing — для автоматической записи данных в реестр, чтобы избежать возможных последствий от системных сбоев или избежать непоследовательности данных о компонентах в случае незавершенного обновления. Фактически, если посмотреть на некоторые файлы транзакций, можно будет увидеть названия разделов, к которым применялись транзакции.

Обслуживанием всех кустов, монтируемых во время загрузки системы, занимается глобальный диспетчер ресурсов реестра — global registry resource manager (RM). RM создается для каждого куста, монтируемого явным образом. Для приложений, использующих реестровые транзакции, создание RM является прозрачной операцией, потому что KTM гарантирует, что все RM-диспетчеры, принимающие участие в одной и той же транзакции, координируются в двухэтапном протоколе фиксации-отмены. Для глобальных реестровых RM файлы регистрации CLFS хранятся, как уже ранее упоминалось, в каталоге System32\Config\Txr. Для других кустов они хранятся рядом с кустом (в том же каталоге). Эти файлы невидимы и следуют тому же самому соглашению об именах, завершаясь расширением .regtrans-ms. Имена этих регистрационных файлов используют в качестве префиксов имена кустов, к которым они относятся.

Отслеживание активности реестра

Поскольку система и приложения в своем поведении слишком сильно зависят от настроек конфигурации, сбои системы и приложения могут возникать из-за изменений данных реестра или установок систем безопасности. Когда система или приложение не может прочитать настройки, считающиеся постоянно доступными, они могут работать неправильно, выводить сообщения об ошибках, не раскрывающие их истинную причину, или даже входить в аварийный режим. Без понимания того, как именно система или приложение не смогло получить доступ к реестру, невозможно разобраться в том, какой из разделов или параметров реестра неправильно настроен. В такой ситуации ответ может дать утилита Process Monitor из инструментария Windows Sysinternals (http://technet.microsoft.com/sysinternals).

Process Monitor позволяет отслеживать активность реестра по мере ее проявления. Для каждого обращения к реестру Process Monitor показывает процессы, осуществляющие доступ, время, тип и результат доступа, а также стек потока и момент обращения. Эту информацию можно использовать для того, чтобы видеть, как приложения и система зависят от реестра, выясняя, где приложения и система хранят настройки конфигурации, и выявляя проблемы, связанные с приложениями, утратившими доступ к разделам или параметрам реестра. Process Monitor включает усовершенствованную фильтрацию и выделение, чтобы можно было раскрыть активность, связанную с определенными разделами или параметрами, или обратиться к активности конкретных процессов.

Внутренние особенности Process Monitor

Работа Process Monitor зависит от драйвера устройства, который в процессе выполнения извлекается из его исполняемого образа, а затем запускается. При первом выполнении этой программы требуется, чтобы учетная запись, с которой она запускается, имела привилегию загрузки драйвера — Load Driver, а также привилегию отладки — Debug. Последующие выполнения в том же сеансе загрузки системы требуют только привилегию отладки, поскольку, будучи однажды загруженным, драйвер остается в качестве резидентного кода.

ЭКСПЕРИМЕНТ: ПРОСМОТР АКТИВНОСТИ РЕЕСТРА НА ПРОСТАИВАЮЩЕЙ СИСТЕМЕ

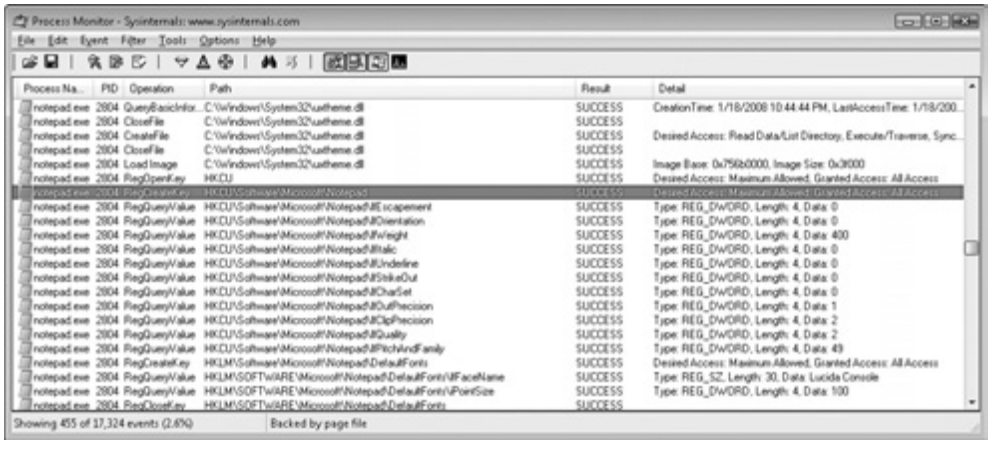
Поскольку в реестре реализована функция регистрационного уведомления об изменении раздела — RegNotifyChangeKey, которую приложения могут использовать для запроса уведомления об изменении реестра без организации его опроса, когда вы запускаете Process Monitor на простаивающей системе, вы можете не увидеть повторяющихся обращений к одним и тем же разделам или параметрам реестра. Любая подобная активность служит признаком плохо написанного приложения, которая безо всяких на то причин негативно влияет на общую производительность системы.

Запустите Process Monitor и через несколько секунд проверьте выходной журнал, чтобы посмотреть, можно ли обнаружить состояние опроса. Щелкните правой кнопкой мыши на строке вывода, связанной с опросом, и затем выберите в контекстном меню пункт Process Properties (Свойство процесса), чтобы просмотреть подробности процесса, проявляющего активность. ■

ЭКСПЕРИМЕНТ: ИСПОЛЬЗОВАНИЕ PROCESS MONITOR ДЛЯ ОПРЕДЕЛЕНИЯ МЕСТОНАХОЖДЕНИЯ НАСТРОЕК ПРИЛОЖЕНИЯ В РЕЕСТРЕ

В некоторых ситуациях, когда нужно выявить причину сбоев, может потребоваться определить, где в реестре система или приложение хранит конкретные настройки. В этом эксперименте для обнаружения настроек программы Блокнот (Notepad) используется утилита Process Monitor. Блокнот, как и большинство приложений Windows, сохраняет от выполнения к выполнению такие пользовательские предпочтения, как режим автоматического переноса слов, используемый шрифт и его размер, а также положение окна. Наблюдая через Process Monitor за чтением или записью программой Блокнот ее настроек, можно определить раздел реестра, в котором эти настройки сохраняются. Для этого нужно выполнить следующие действия:

1. Заставьте Блокнот сохранить такую настройку, которую можно легко найти в данных, отслеживаемых Process Monitor. Это можно сделать, если запустить Блокнот, установить шрифт Times New Roman, а затем выйти из Блокнота.
2. Запустите Process Monitor. Откройте диалоговое окно фильтра и фильтр имени процесса — Process Name, а затем наберите в качестве строки соответствия notepad.exe. Это действие уточнит, что Process Monitor будет регистрировать только активность, проявляемую процессом notepad.exe.
3. Запустите Блокнот еще раз, и после того как он запустится, остановите имеющийся в Process Monitor захват события, переключив состояние Capture Events (Захват событий) в меню File (Файл).
4. Прокрутите получившийся журнал к его самой верхней записи и выберите эту запись.
5. Нажмите сочетание клавиш Ctrl+F, чтобы открыть диалоговое окно Find (Поиск), и задайте поиск строки times new. Process Monitor должен выделить строку, похожую на ту, что изображена на следующем экране, которая представляет собой Блокнот, который читает значение шрифта из реестра. Другие операции, показанные в непосредственной близости, должны относиться к другим настройкам программы Блокнот.



6. В завершение щелкните правой кнопкой мыши на выделенной строке и щелкните на пункте Jump To (Перейти к). Process Monitor запустит редактор реестра Regedit (если он до этого не работал) и заставит его перейти к параметру реестра, относящемуся к программе Блокнот, и выбрать этот параметр. ■

Технологии поиска и устранения неисправностей с помощью Process Monitor

Для понимания проблем приложения или системы, связанных с реестром, можно применить следующие две основные технологии поиска и устранения неисправностей, имеющиеся в Process Monitor.

- Просмотр последнего следа активности приложения в Process Monitor, который был до сбоя. Это действие может указать на проблему.
- Сравнивание следа в Process Monitor, оставленного сбойным приложением, со следом из работающей системы.

Чтобы использовать первый подход, запустите Process Monitor, а затем запустите приложение. В том месте, где произошел сбой, вернитесь к Process Monitor и остановите регистрацию (нажав комбинацию клавиш Ctrl+E). Затем перейдите к концу журнала и найдите последние операции, выполнявшиеся приложением перед сбоем, аварией, зависанием или чем-нибудь еще. Начиная с последней строки, вернитесь к прежним событиям, проверяя файлы, разделы реестра (или и то и другое), на которые делались ссылки — зачастую это должно помочь засечь источник проблемы.

Если приложение дало сбой на одной системе, но работает на другой, следует воспользоваться вторым подходом. Захватите следы приложения в Process Monitor на работающей и сбойной системе и сохраните вывод в регистрационном файле. Затем откройте регистрационные файлы из работающей и из сбойной системы с помощью Microsoft Excel (взяв за основу умолчания в мастере импортирования) и удалите первые три столбца[1]. И наконец, сравните получившиеся файлы регистрации. (Вы можете сделать это с помощью утилиты WinDiff, входящей в состав Windows SDK.)

Записи в трассировке Process Monitor, имеющие в столбце Result значения NAME NOT FOUND (имя не найдено) или ACCESS DENIED (доступ запрещен), и являются теми записями, которые нужно изучить. Запись со значением NAME NOT FOUND появляется, когда приложение пытается прочитать данные из несуществующего раздела или параметра реестра. Во многих случаях ненайденный раздел или параметр не создает никаких проблем, поскольку процесс, потерпевший неудачу при чтении настроек из реестра, просто возвращается к значениям по умолчанию. Но в некоторых случаях приложения рассчитывают найти параметры, для которых нет значений по умолчанию, и дадут сбой, если эти параметры не будут найдены.

Ошибки запрещения доступа чаще всего происходят из-за сбоев приложения, связанного с реестром, и случаются, когда приложение не имеет разрешения на

[1] Если не удалить первые три столбца, сравнение будет показывать каждую строку как содержащую отличия, поскольку в первых трех столбцах содержится информация, отличающаяся от запуска к запуску, такая как время и идентификатор процесса.

доступ к разделу реестра тем путем, которое ему требуется. Приложения, не проверяющие результаты операций с реестром или не проводящие должного восстановления после ошибок, дадут сбой.

Чаще всего вызывает опасения результирующая строка BUFFER OVERFLOW (переполнение буфера). Она не является свидетельством того, что переполнение буфера произошло именно в приложении, которое ее получило. Напротив, она используется диспетчером конфигурации, чтобы проинформировать приложение о том, что буфер, определенный для хранения параметра реестра, слишком мал, чтобы вместить этот параметр. Разработчики приложений часто пользуются этим поведением, чтобы определить, какого размера буфер нужно выделить для хранения параметра. Сначала они выполняют запрос с буфером нулевой длины, который возвращает ошибку переполнения буфера и длину данных, которые в него пытались прочитать. Затем приложение выделяет буфер указанного размера и снова считывает параметр. Поэтому вы увидите, что те операции, которые вернули BUFFER OVERFLOW, будут повторены с успешным результатом.

В одном из примеров использования Process Monitor для поиска и устранения реальной проблемы он уберег пользователя от полной переустановки его системы Windows. Симптомом послужило зависание Internet Explorer в момент его запуска, если пользователь сначала не подключился к Интернету. Это интернет-подключение было установлено для системы как подключение по умолчанию, поэтому запуск Internet Explorer должен был вызвать автоматическое телефонное подключение к Интернету (поскольку Internet Explorer был настроен на показ главной страницы, используемой по умолчанию).

Изучение в Process Monitor регистрационного журнала пусковой активности Internet Explorer и возвращение назад из того места в журнале, где завис Internet Explorer, показало запрос раздела HKCU\Software\Microsoft\RAS Phonebook. Пользователь рассказал, что ранее он удалил программу дозвона, связанную с разделом реестра, и создал подключение по телефонной линии вручную. Поскольку имя подключения по телефону не соответствовало тому, что использовалось в удаленной программе дозвона, получилось так, что раздел не был удален при удалении программы дозвона, и это вызвало зависание Internet Explorer. После того как раздел был удален, Internet Explorer стал работать вполне предсказуемым образом.

Регистрационная активность при работе с непривилегированными учетными записями или в процессе входа-выхода из системы

Довольно часто сбои приложений являются результатом того, что приложение работает при запуске с учетной записью, относящейся к административной группе, но не работает при запуске с учетной записью непривилегированного пользователя. Как уже ранее упоминалось, выполняемая программа Process Monitor требует привилегий безопасности, которые обычно не даются стандартным учетным записям пользователей. Но можно проследить работу приложений, выполняемых в сеансе входа в систему непривилегированного пользователя, использовав для выполнения Process Monitor с учетной записью администратора команду Runas.

Если проблемы с реестром связаны с входом в систему с определенной учетной записью или с выходом из нее, вам также нужно предпринять специальные действия, чтобы получить возможность использовать Process Monitor для отслеживания таких этапов. Работа приложений, запущенных с учетной записью локальной системы, не завершаются при выходе пользователя из системы, и этим фактом можно воспользоваться, чтобы заставить Process Monitor работать и после выхода из системы и при последующем входе в нее. Программу Process Monitor можно запустить с учетной записью локальной системы либо используя команду At, которая встроена в Windows, и указания ключа /interactive, либо используя входящую в комплект Sysinternals утилиту PsExec:

```
psexec -i 0 -s -d c:\procmon.exe
```

Ключ -i 0 заставляет PsExec вывести окно программы Process Monitor в сеансе 0 используемого по умолчанию рабочего стола интерактивной станции окна. Ключ -s заставляет PsExec запустить Process Monitor с учетной записью локальной системы, а ключ -d заставляет PsExec запустить Process Monitor и выйти, не дожидаясь завершения работы Process Monitor. При запуске этой команды выполняемый экземпляр Process Monitor переживет выход из системы и снова появится на рабочем столе, когда вы снова войдете в систему, захватывая активность реестра при обоих действиях.

Еще одним способом отслеживания активности реестра при входе в систему и выходе из нее, загрузке системы или завершении процесса, является использование свойства программы Process Monitor по регистрации загрузки. Это свойство можно включить, установив флажок Enable Boot Logging (Включить регистрацию загрузки) в меню Options (Настройки). При следующей загрузке системы драйвер устройства Process Monitor зарегистрирует активность реестра с самого начала загрузки в файле %SystemRoot%\Procmon.pml. Регистрационная информация будет продолжать записываться в этот файл, пока не будет исчерпано дисковое пространство, не будет завершена работа системы или не будет запущена программа Process Monitor. В регистрационном файле сохраняется отслеживание активности при запуске системы, при входе в нее и выходе из нее и при завершении работы системы Windows, что обычно занимает объем от 50 до 150 Мбайт.

Внутреннее устройство реестра

В данном разделе вы узнаете, как диспетчер конфигурации — подсистема исполняющей системы, в которой реализован реестр, — организует файлы реестра на диске. Будет изучен порядок, используемый диспетчером конфигурации для управления реестром, по мере того как приложения и другие компоненты операционной системы считывают и изменяют разделы и параметры реестра. Будет также рассмотрен механизм, с помощью которого диспетчер конфигурации пытается обеспечить нахождение реестра в неизменно восстанавливаемом состоянии даже при сбое системы в ходе внесения изменений в реестр.

Кусты

На диске реестр не является обычным большим файлом, а представляет собой набор отдельных файлов, которые называются кустами. Каждый куст содержит

дерево реестра, у которого есть раздел, служащий ему корнем или отправной точкой дерева. Подразделы и их параметры находятся ниже корня. Можно подумать, что корневые разделы, отображаемые в редакторе реестра, соответствуют корневым разделам в кустах, но так бывает не всегда. В табл. 4.5 перечислены кусты реестра и имена их файлов при хранении на диске. Путевые имена всех кустов, за исключением тех, которые используются для профилей пользователей, кодируются в диспетчере конфигурации. По мере того как диспетчер конфигурации загружает кусты, включая профили системы, он записывает путь к каждому кусту в параметрах подраздела HKLM\SYSTEM\CurrentControlSet\Control\Hivelist, удаляя путь при выгрузке куста. Он создает корневые разделы, связывает эти кусты вместе, чтобы построить структуру реестра, с которой вы знакомы и которая показывается редактором реестра.

Вы заметите, что некоторые из этих кустов, перечисленные в табл. 4.5, могут изменяться и не имеют связанных с ними файлов. Система создает эти кусты и управляет ими целиком в памяти, поэтому такие кусты являются временными. Система создает непостоянные кусты при каждой своей загрузке. В качестве примера непостоянного куста можно привести HKLM\HARDWARE, в котором хранится информация о физических устройствах и выделенных этим устройствам ресурсах. Выделение ресурсов и определение установленного оборудования проводятся при каждой загрузке системы, поэтому хранить эти данные на диске было бы нелогично.

Таблица 4.5. Соответствие файлов на диске путям в реестре

Путь куста реестра	Путь файла куста
HKEY_LOCAL_MACHINE\BCD00000000	\Boot\BCD
HKEY_LOCAL_MACHINE\COMPONENTS	%SystemRoot%\System32\Config\Components
HKEY_LOCAL_MACHINE\SYSTEM	%SystemRoot%\System32\Config\System
HKEY_LOCAL_MACHINE\SAM	%SystemRoot%\System32\Config\Sam
HKEY_LOCAL_MACHINE\SECURITY	%SystemRoot%\System32\Config\Security
HKEY_LOCAL_MACHINE\SOFTWARE	%SystemRoot%\System32\Config\Software
HKEY_LOCAL_MACHINE\HARDWARE	Непостоянный куст
HKEY_USERS\<*SID учетной записи локальной службы*>	%SystemRoot%\ServiceProfiles\LocalService\Ntuser.dat
HKEY_USERS\<*SID учетной записи сетевой службы*>	%SystemRoot%\ServiceProfiles\NetworkService\NtUser.dat
HKEY_USERS\<*SID имени пользователя*>	\Users\<username>\Ntuser.dat
HKEY_USERS\<*SID имени пользователя*>_Classes	\Users\<username>\AppData\Local\Microsoft\Windows\Usrclass.dat
HKEY_USERS\.DEFAULT	%SystemRoot%\System32\Config\Default

Ограничения размера куста

В некоторых случаях размеры куста ограничиваются. Например, Windows накладывает ограничения на размер куста HKLM\SYSTEM. Она поступает таким

образом, потому что Winload считывает весь куст HKLM\SYSTEM в физическую память практически сразу же после запуска процесса загрузки, когда разбиение на страницы виртуальной памяти еще не включено. Программа Winload также загружает в физическую память Ntoskrnl и драйверы устройств загрузки, поэтому она должна ограничивать физическую память, выделяемую HKLM\SYSTEM. На 32-разрядных системах Winload позволяет кусту быть размером до 400 Мбайт или размером в половину объема физической памяти системы, в зависимости от того, какой из размеров меньше. На системах x64 нижняя граница составляет 1,5 Гбайт. На системах Itanium она составляет 32 Мбайт.

ЭКСПЕРИМЕНТ: ЗАГРУЗКА И ВЫГРУЗКА КУСТОВ ВРУЧНУЮ

У программы Regedit есть возможность загружать кусты, к которым можно получить доступ через ее меню Файл (File). Эта возможность может пригодиться при поиске и устранении неисправностей, когда нужно просмотреть или отредактировать куст из незагружаемой системы или носителя резервной копии. В этом эксперименте Regedit будет использоваться для загрузки версии куста HKLM\SYSTEM, который программа Windows Setup создает в процессе установки.

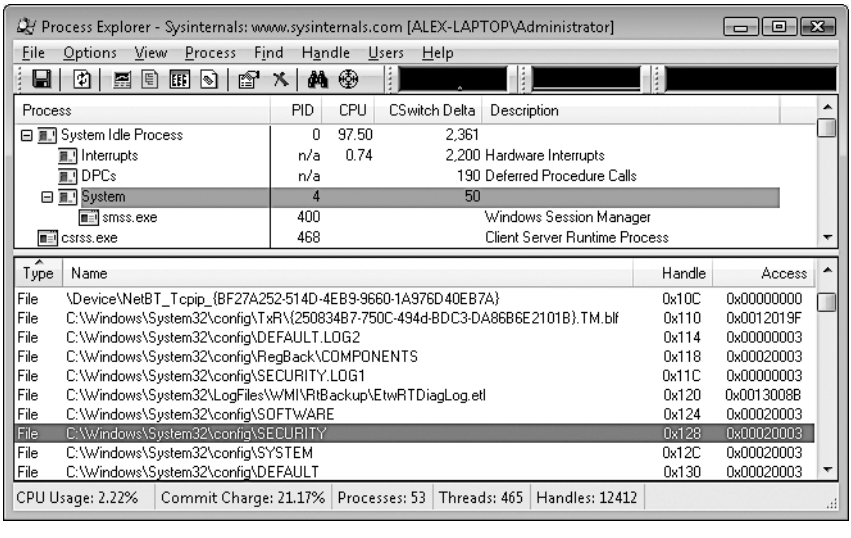
1. Кусты могут быть загружены только ниже HKLM или HKU, поэтому откройте Regedit, выберите HKLM и выберите пункт Загрузить куст (Load Hive) из меню Regedit Файл (File).
2. В диалоговом окне Загрузить куст (Load Hive) перейдите в каталог %SystemRoot%\System32\Config\RegBack, выберите файл System и откройте его. По запросу на ввод наберите Test в качестве имени раздела, куда он будет загружен.
3. Откройте только что созданный раздел HKLM\Test и исследуйте содержимое этого куста.
4. Откройте HKLM\SYSTEM\CurrentControlSet\Control\Hivelist и найдите запись \Registry\Machine\Test, которая показывает, как диспетчер конфигурации управляет списками загруженных кустов в разделе Hivelist.
5. Выберите раздел HKLM\Test, а затем выберите из меню Файл (File) программы Regedit пункт Выгрузить куст (Unload Hive). ■

Символические ссылки реестра

Специальный тип раздела, известный как символические ссылки реестра, позволяет диспетчеру конфигурации связывать разделы с целю организации реестра. Символическая ссылка является разделом, перенаправляющим диспетчер конфигурации на другой раздел. Таким образом, раздел HKLM\SAM является символической ссылкой на раздел в корне куста SAM. Символические ссылки создаются путем указания функции RegCreateKey или функции RegCreateKeyEx параметра REG_CREATE_LINK. Внутри диспетчер конфигурации создаст параметр REG_LINK, называемый SymbolicLinkValue, который будет содержать путь к целевому разделу. Поскольку этот параметр относится к типу REG_LINK, а не к типу REG_SZ, он не будет виден в Regedit, но тем не менее он будет частью куста реестра, хранящегося на диске.

ЭКСПЕРИМЕНТ: ПРОСМОТР ДЕСКРИПТОРОВ КУСТОВ

Диспетчер конфигурации открывает кусты, используя таблицу дескрипторов ядра (рассматриваемую в главе 3), так что он может получить доступ к кустам из любого контекста процесса. Использование таблиц дескрипторов ядра является хорошей альтернативой подходу, использующему драйверы или компоненты исполняющей системы для получения доступа из процесса System только к тем дескрипторам, которые должны быть защищены от пользовательских процессов. Чтобы посмотреть на дескрипторы кустов, можно воспользоваться программой Process Explorer, в которой кусты будут отображаться по мере своего открытия в процессе System. Выберите процесс System, а затем установите флажок Handles (Дескрипторы) в пункте меню Lower Pane View (Вид нижней панели), в который можно попасть из меню View (Вид). Отсортируйте список по типу дескриптора и прокручивайте список до тех пор, пока не увидите файлы кустов.



Структура куста

Диспетчер конфигурации логически делит куст на распределяемые единицы, называемые блоками, способом, похожим на то, как файловая система делит диск на кластеры. По определению, размер блока реестра составляет 4096 байт (4 Кб). Когда куст расширяется за счет новых данных, он всегда расширяется за счет увеличения количества блоков. Первый блок куста называется базовым блоком.

Базовый блок включает в себя глобальную информацию о кусте, в которую входят:

- сигнатура regf, которая идентифицирует файл как куст;
- обновляемые последовательные номера;
- отметка времени, показывающая, когда в последний раз в отношении куста применялась операция записи[1];

[1] Значение обновляемой последовательности номеров и отметок времени будет объяснено при рассмотрении записи данных в файл куста.

- информация о ремонте или восстановлении реестра, производимом Winload;
- номер версии формата куста;
- контрольная сумма;
- внутреннее файловое имя файла куста (например, \Device\HarddiskVolume1\WINDOWS\SYSTEM32\CONFIG\SAM).

Номер версии формата куста определяет формат данных внутри куста. Диспетчер конфигурации использует формат куста версии 1.3 (с усовершенствованным поиском за счет кэширования первых четырех символов имени внутри ячеистой структуры индекса) для всех кустов, за исключением System и Software, для совместимости по перемещаемым профилям с Windows 2000. Для кустов System и Software он использует версию 1.5 из-за более поздних оптимизаций формата для объемных параметров (более 1 Мбайт) и поиска (вместо кэширования первых четырех символов имени для уменьшения конфликтных ситуаций используется хэш полного имени).

Windows упорядочивает данные реестра, которые куст хранит в контейнерах, называемых ячейками. В ячейке может храниться раздел, параметр, дескриптор безопасности, список подразделов или список параметров раздела. Четырехбайтовый символьный тег в начале данных ячейки описывает тип данных в виде сигнатуры. Подробное описание типа данных ячейки дается в табл. 4.6. Заголовком ячейки является поле, которое в виде единичного дополнения (не представленного в структурах CM_) определяет размер ячейки. Когда ячейка присоединяется к кусту, и куст должен быть расширен, чтобы ее вместить, система создает единичный блок, который называется приемником (bin).

Таблица 4.6. Типы данных ячейки

Тип данных	Тип структуры	Описание
Ячейка раздела	CM_KEY_NODE	Ячейка, содержащая раздел реестра, которая также называется узлом раздела (key node). В ячейке раздела содержатся: • сигнатура (kn для раздела, kl для узла ссылки); • отметка времени самого последнего обновления раздела; • индекс ячейки, в которой содержится родительский раздел данного раздела; • индекс ячейки списка подразделов, идентифицирующего подразделы данного раздела; • индекс ячейки дескриптора безопасности данного раздела; • индекс ячейки строки, определяющей имя класса данного раздела; • имя раздела (например, CurrentControlSet). В ней также хранится кэшированная информация, например количество подразделов данного раздела, а также размер самого большого раздела, имя параметра, данные параметра и имя класса подразделов данного раздела

Тип данных	Тип структуры	Описание
Ячейка параметра	CM_KEY_ VALUE	Ячейка, содержащая информацию о параметре раздела. Эта ячейка включает сигнатуру (kv), тип параметра (например, REG_ DWORD или REG_BINARY) и имя параметра (например, Boot-Execute). Ячейка параметра содержит также индекс той ячейки, в которой содержатся данные параметра
Ячейка списка подразделов	CM_KEY_ INDEX	Ячейка, состоящая из списка индексов ячеек разделов, представляющих собой подразделы общего родительского раздела
Ячейка списка параметров	CM_KEY_ INDEX	Ячейка, состоящая из списка индексов ячеек параметров, представляющих собой параметры общего родительского раздела
Ячейка дескриптора безопасности	CM_KEY_ SECURITY	Ячейка, содержащая дескриптор безопасности. Ячейки дескрипторов безопасности включают сигнатуру (ks), помещаемую в заголовок ячейки, и счетчик ссылок, в который записывается количество узлов раздела, совместно использующих дескриптор безопасности. Ячейки дескриптора безопасности могут совместно использоваться несколькими ячейками раздела

Приемник — это размер новой ячейки, округленный вверх до следующей границы блока или страницы, в зависимости от того, что выше. Система рассматривает любое пространство между окончанием ячейки и окончанием приемника в качестве свободного пространства, которое она может выделить другим ячейкам. У приемников также есть заголовки, в которых содержится сигнатура, hbin и поле, в которое записывается смещение приемника в файле куста и размер приемника.

Windows минимизирует ряд рутинных операций управления путем использования приемников вместо ячеек для отслеживания активных частей реестра. Например, система обычно назначает и освобождает приемники реже, чем ячейки, что позволяет диспетчеру конфигурации более эффективно считывать куст реестра в память. Когда диспетчер конфигурации считывает куст в память, он читает весь куст, включая пустые приемники, но позже он может их отбросить. Когда система добавляет ячейки в куст и удаляет их оттуда, куст может содержать пустые приемники, перемежающиеся активными приемниками. Эта ситуация похожа на фрагментацию диска, которая происходит, когда система создает и удаляет файлы на диске. Когда приемник становится пустым, диспетчер конфигурации присоединяет к пустому приемнику любые примыкающие пустые приемники для формирования как можно более крупного и непрерывного пустого приемника. Диспетчер конфигурации также присоединяет друг к другу примыкающие удаленные ячейки для формирования более крупных свободных ячеек[1].

Ссылки, образующие структуру куста, называются индексами ячеек. Индекс ячейки является смещением ячейки в файле куста за вычетом размера базового

[1] Диспетчер конфигурации уменьшает куст только тогда, когда становятся свободными приемники в конце куста. Вы можете сделать реестр компактнее путем создания его резервной копии и его восстановления, используя Windows-функции RegSaveKey и RegReplaceKey, которые используются утилитой Windows Backup.

блока. Таким образом, индекс ячейки похож на указатель из одной ячейки на другую ячейку, который диспетчер конфигурации интерпретирует относительно начала куста. Например, как видно из табл. 4.6, ячейка, описывающая раздел, содержит поле, определяющее индекс ячейки родительского раздела и индекс ячейки для подразделов, определяющей ячейку, описывающую подразделы, подчиненные указанному подразделу. Ячейка со списком подразделов содержит список индексов ячеек, относящихся к ячейкам разделов данного подраздела. Следовательно, если нужно найти, скажем, ячейку раздела подраздела А, родителем которого является раздел Б, нужно сначала найти ячейку, содержащую список подразделов раздела Б, используя индекс ячейки списка подразделов в ячейке раздела Б. Затем с помощью списка индексов ячеек в ячейке списка подразделов определяется местонахождение ячеек каждого подраздела, подчиненного разделу Б. В каждой ячейке подраздела проводится проверка, не совпадает ли имя подраздела, хранящееся в ячейке раздела, с именем того подраздела, который нужно найти, в данном случае с именем подраздела А.

В том, чем отличаются друг от друга ячейки и приемники, нетрудно запутаться. Поэтому, чтобы разобраться в различиях, давайте посмотрим на пример плана простого куста реестра. Приводимый в качестве примера на рис. 4.3 файл куста реестра содержит базовый блок и два приемника. Первый приемник пустой, а второй приемник содержит несколько ячеек. Логически, у куста имеются только два раздела: Root и его подраздел Sub Key. У раздела Root имеются два параметра, Val 1 и Val 2. Ячейка списка подразделов определяет местонахождение подраздела, подчиненного разделу Root, а ячейка списка параметров определяет местонахождение параметров раздела Root. Пустые места во втором приемнике являются пустыми ячейками. На рис. 4.3 не показаны ячейки безопасности для двух разделов, которые присутствовали бы в кусте.

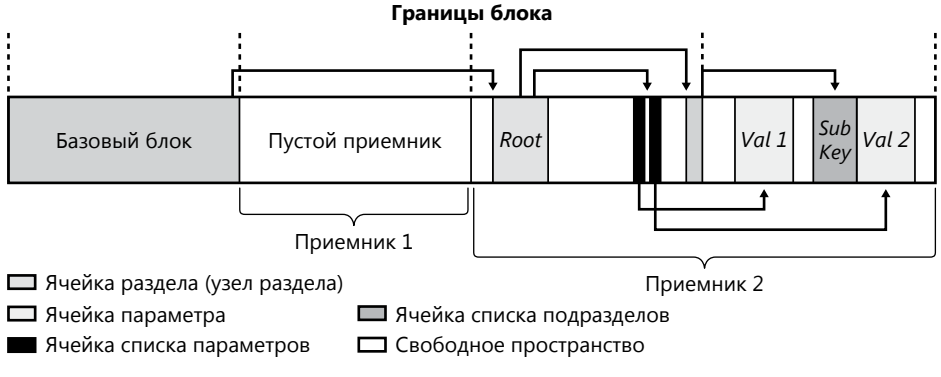


Рис. 4.3. Внутренняя структура куста реестра

Для оптимизации поисков параметров и подразделов диспетчер конфигурации сортирует ячейки списка подразделов в алфавитном порядке. Затем диспетчер конфигурации может провести двоичный поиск для того, чтобы найти в списке подразделов нужный подраздел. Диспетчер конфигурации проверяет подраздел в середине списка, и если имя подраздела, который разыскивается диспетчером конфигурации, располагается по алфавиту перед именем среднего подраздела,

диспетчер конфигурации знает, что подраздел находится в первой половине списка подразделов; в противном случае подраздел находится во второй половине списка подразделов. Этот процесс разбиения продолжается до тех пор, пока диспетчер конфигурации не найдет подраздел или не найдет никаких совпадений. А вот ячейки списка параметров не сортируются, поэтому новые значения всегда добавляются к концу списка.

Отображения ячеек

Если кусты не растут, диспетчер конфигурации может выполнить всю свою работу по управлению реестром в той версии куста, которая содержится в памяти, как будто куст является файлом. Благодаря индексу ячейки диспетчер конфигурации может вычислить местонахождение ячейки в памяти путем простого прибавления индекса ячейки, который является смещением в файле куста, к базе образа куста в памяти. В самом начале загрузки системы именно этим и занимается Winload с кустом SYSTEM: Winload считывает весь куст SYSTEM в память в качестве куста, доступного только для чтения, и прибавляет индексы ячеек к базе образа куста в памяти для определения местонахождения ячеек. К сожалению, кусты разрастаются по мере добавления новых разделов и параметров, стало быть, система должна выделять пулы выгружаемой памяти для хранения новых приемников, содержащих добавленные разделы и параметры. Следовательно, пул выгружаемой памяти, содержащий данные реестра в памяти, не обязательно должен быть непрерывным.

Для работы с непоследовательными адресами памяти, ссылающимися на данные куста в памяти, диспетчер конфигурации заимствует стратегию, подобную той, что используется диспетчером памяти Windows для отображения адресов виртуальной памяти на адреса физической. Диспетчер конфигурации использует двухуровневую схему, показанную на рис. 4.4, в которой в качестве ввода берется индекс ячейки (то есть смещение в файле куста), а возвращается в качестве вывода как адрес в памяти того блока, который занят индексом ячейки, так и адрес в памяти того блока, который занят ячейкой. Следует помнить, что приемник может содержать один или несколько блоков и что кусты прирастают количеством приемников, поэтому Windows всегда представляет приемник в виде непрерывной области памяти. Поэтому все блоки внутри приемника оказываются внутри одного и того же представления диспетчера кэша.

Для реализации отображения диспетчер конфигурации логически делит индекс ячейки на два поля, точно так же, как диспетчер памяти делит на поля виртуальный адрес. Windows интерпретирует первое поле индекса ячейки, как индекс в каталоге отображения ячеек, принадлежащем кусту. Каталог отображения ячеек содержит 1024 записи, каждая из которых ссылается на таблицу отображения ячеек, состоящую из 512 записей отображений. Запись в этой таблице отображения ячеек определяется вторым полем в индексе ячейки. Эта запись определяет адреса приемника и блока памяти ячейки. Не все приемники обязательно проецируются на память, и если при поиске ячейки выдается адрес 0, диспетчер конфигурации отображает приемник в памяти, убирая, если нужно, отображение другого приемника в обслуживаемом этим диспетчером LRU-списке отображения.

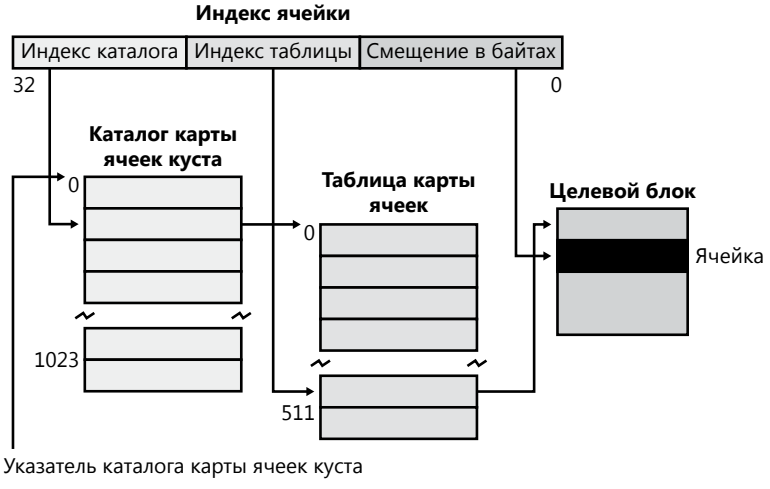


Рис. 4.4. Структура индекса ячейки

ЭКСПЕРИМЕНТ: НАБЛЮДЕНИЕ ЗА ИСПОЛЬЗОВАНИЕМ ПУЛА ВЫГРУЖАЕМОЙ ПАМЯТИ КУСТА

Инструментария, работающего на административном уровне, показывающего количество пулов выгружаемой памяти, расходуемой в Windows кустами реестра, включая профили пользователей, не существует. Тем не менее команда отладчика ядра !reg dumppool показывает не только то количество страниц выгружаемой памяти, которое занято каждым загруженным кустом, но также и то количество страниц, в котором хранятся временные и долговременные данные. В конце своего вывода команда показывает суммарные данные о занятой кустами памяти. (Команда показывает только последние 32 символа имени куста.)

```
kd> !reg dumppool

dumping hive at e20d66a8 (a\Microsoft\Windows\UsrClass.dat)
  Stable Length = 1000
  1/1 pages present
  Volatile Length = 0

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
```

■

На завершающем этапе процесса трансляции диспетчер конфигурации интерпретирует последнее поле индекса ячейки как смещение в идентифициро-

ванном блоке для точного определения местоположения ячейки в памяти. При инициализации куста диспетчер конфигурации создает в динамическом режиме таблицы отображения, определяя запись отображения для каждого блока в кусте, и добавляет и удаляет таблицы из каталога ячеек по мере востребованности изменений размера куста.

Пространство имен и работа реестра

Для интеграции пространства имен реестра с общим пространством имен ядра диспетчер конфигурации определяет тип объекта «раздел». Диспетчер конфигурации вставляет объект типа раздел Registry в корень пространства имен Windows, и он служит в качестве точки входа в реестр. Программа Regedit показывает имена разделов в форме HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet, но подсистема Windows переводит такие имена в форму своего пространства имен объектов (например, \Registry\Machine\System\CurrentControlSet). Когда диспетчер объектов Windows проводит разбор этого имени, он сначала определяет объект раздела Registry и вручает остальную часть имени диспетчеру конфигурации. Диспетчер конфигурации принимает на себя разбор имени, проводя поиск по своему внутреннему дереву куста, чтобы найти желаемый раздел или параметр. Перед тем как давать описание передачи управления обычной операции с реестром, нужно рассмотреть объекты разделов и блоки управления разделов. Когда приложение открывает или создает раздел реестра, диспетчер объектов дает дескриптор, с помощью которого приложение ссылается на раздел. Дескриптор соответствует объекту раздела, который назначается диспетчером конфигурации с помощью диспетчера объектов. Используя поддержку объектов со стороны диспетчера объектов, диспетчер конфигурации пользуется функциональными возможностями по обеспечению безопасности и подсчетом ссылок, которые предоставляются диспетчером объектов.

Для каждого открытого раздела реестра диспетчер конфигурации также назначает блок управления разделом. В блоке управления разделом хранится имя раздела, в него же включается индекс ячейки узла раздела, на который ссылается блок управления, и содержится флаг, который отмечает, нужно ли диспетчеру конфигурации удалить ячейку раздела, на которую ссылается блок управления разделом, когда закрывается последний дескриптор раздела. Windows помещает все блоки управления разделами в хэш-таблицу, чтобы дать возможность быстрого поиска существующих блоков управления разделами по имени. Объект раздела указывает на соответствующий ему блок управления разделом, поэтому если два приложения открывают один и тот же раздел реестра, каждое из них получает объект раздела, и оба объекта раздела указывают на общий блок управления разделом.

Когда приложение открывает существующий раздел реестра, управление сначала передается в API-функцию реестра вместе с указанным приложением именем раздела реестра, а эта функция вызывает процедуру разбора имени диспетчера объектов. Диспетчер объектов, как только встретит в пространстве имен принадлежащий диспетчеру конфигурации объект раздела реестра, передает путевое имя диспетчеру конфигурации. Диспетчер конфигурации выполняет поиск хэш-таблицы блоков управления разделами. Если связанный блок управления разделом будет там найден, дополнительная работа не потребуется, в противном

случае поиск предоставляет диспетчеру конфигурации ближайший к искомому разделу блок управления разделом, и поиск продолжается с использованием структуры данных куста, находящегося в памяти, чтобы провести поиск среди разделов и подразделов с целью найти указанный раздел. Если диспетчер конфигурации находит ячейку раздела, он проводит поиск в дереве блока управления раздела, чтобы определить, открыт ли раздел (этим же или каким-нибудь другим приложением). Процедура поиска оптимизирована таким образом, чтобы всегда начинать с самого близкого предшественника с уже открытым блоком управления разделом. Например, если приложение открывает \Registry\Machine\Key1\Subkey2 и \Registry\Machine уже открыт, процедура разбора использует в качестве начальной точки блок управления разделом \Registry\Machine. Если раздел открыт, диспетчер конфигурации увеличивает показание счетчика ссылок на существующий блок управления разделом. Если раздел не открыт, диспетчер конфигурации назначает новый блок управления разделом и вставляет его в дерево. Затем диспетчер конфигурации назначает объект раздела, указывает на объект раздела в блоке управления разделом и возвращает управление диспетчеру объектов, которые возвращает дескриптор приложению.

Когда приложение создает новый раздел реестра, диспетчер конфигурации сначала находит ячейку раздела родителя нового раздела. Затем диспетчер конфигурации просматривает список свободных ячеек для того куста, в котором будет размещаться новый раздел, чтобы определить, существуют ли ячейки, достаточно большие, чтобы содержать ячейку нового раздела. Если нет каких-либо достаточно больших свободных ячеек, диспетчер конфигурации назначает новый приемник и использует его для ячейки, размещая любое пространство в конце приемника в списке свободных ячеек. Новая ячейка раздела заполняется относящейся к делу информацией, включая имя раздела, и диспетчер конфигурации добавляет ячейку раздела к списку подразделов ячейки списка подразделов, принадлежащего родительскому разделу. И наконец, система сохраняет индекс родительской ячейки в ячейке нового подраздела.

Диспетчер конфигурации использует счетчик ссылок блока управления разделом, чтобы определить, когда удалять этот блок. Когда все дескрипторы, ссылающиеся на раздел в блоке управления разделом, закрываются, счетчик ссылок приобретает значение 0, что указывает на то, что блок управления разделом больше не нужен. Если приложение, которое вызывает API-функцию для удаления раздела, устанавливает флаг удаления, диспетчер конфигурации может удалить соответствующий раздел из куста разделов, поскольку он знает, что ни одно из приложений не держит раздел открытым.

ЭКСПЕРИМЕНТ: ПРОСМОТР БЛОКОВ УПРАВЛЕНИЯ РАЗДЕЛАМИ

Для вывода списка всех блоков управления разделами, назначенных системой, можно воспользоваться командой !reg openkeys. Кроме того, если нужно просмотреть блок управления разделом конкретного открытого раздела, можно воспользоваться командой !reg findkcb:

```
kd> !reg findkcb \registry\machine\software\microsoft

Found KCB = e1034d40 :: \REGISTRY\MACHINE\SOFTWARE\MICROSOFT
You can then examine a reported key control block with the !reg kcb command:
```

```
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 показывает, что имя хранится в сжатом виде, а поле SubKeyCount показывает, что у раздела есть 137 подразделов. ■

Обеспечение надежного хранения

Для обеспечения постоянной возможности восстановления долговременных кустов реестра (у которых имеются файлы на диске), диспетчер конфигурации использует регистрационные кусты (log hives). У каждого долговременного куста есть связанный с ним регистрационный куст, являющийся скрытым файлом с тем же базовым именем, что и у куста, и с расширением logN. Для обеспечения прогресса диспетчер конфигурации использует схему двойной регистрации. Потенциально существует два регистрационных файла: .log1 и .log2. Если по каким-то причинам .log1 был записан, но при записи изменившихся данных в первичный регистрационный файл произошел сбой, то при следующем сбросе произойдет переключение на .log2 с совокупными изменившимися данными. Если при этой записи также произойдет сбой, совокупные изменившиеся данные (данные в .log1 и данные, изменившиеся в промежутке) сохраняются в .log2. Как следствие, .log1 будет использован снова и в следующий раз, пока не будет успешно осуществлена операция записи в первичный файл регистрации. Если сбоя не будет, используется только .log1.

Например, если заглянуть в ваш каталог %SystemRoot%\System32\Config, можно увидеть System.log1, Sam.log1 и другие файлы с расширениями .log1 и .log2. При инициализации куста диспетчер конфигурации назначает битовый массив, в котором каждый бит представляет 512-байтную порцию, или сектор куста. Этот массив называется массивом изменившихся секторов, потому что установленный бит в массиве означает, что система изменила соответствующий сектор куста в памяти и должна записать сектор обратно в файл куста.

(Сброшенный бит означает, что соответствующий сектор не устарел по сравнению с содержимым куста, находящегося в памяти.)

При создании нового или изменении существующего раздела или параметра диспетчер конфигурации отмечает изменившиеся сектора куста в принадлежащем кусту массиве изменившихся секторов. Затем диспетчер конфигурации планирует отложенную операцию записи или синхронизацию куста. Системный поток отложенной записи куста просыпается через пять секунд после запроса на синхронизацию куста и записывает изменившиеся сектора куста для всех кустов из памяти в файлы кустов на диске. Таким образом, система сбрасывает на диск и все изменения реестра, которые произошли между запросом на синхронизацию куста и самой синхронизацией. После синхронизации куста следующая синхронизация произойдет не ранее, чем через пять секунд.

ПРИМЕЧАНИЕ

Имя API-функции RegFlushKey подразумевает, что функция сбрасывает на диск только измененные данные указанного раздела, но на самом деле она инициирует полный сброс реестра, что сильно влияет на производительность системы. По этой причине и с учетом того факта, что реестр автоматически обеспечивает сброс измененных данных в надежное хранилище в течение нескольких секунд, прикладные программисты должны избегать использования этой функции.

Если программа отложенной записи просто записала все изменившиеся сектора в файл куста и система дала сбой в середине операции, файл куста не будет соответствовать изменениям и поддаваться восстановлению. Чтобы исключить подобную ситуацию, система отложенной записи сначала сбрасывает массив изменившихся секторов куста и все изменившиеся сектора в файл регистрации куста, увеличивая, если это необходимо, размер регистрационного файла. Затем система отложенной записи обновляет порядковый номер в базовом блоке куста и записывает изменившиеся сектора в куст. Когда система отложенной записи завершит свою работу, она обновляет второй порядковый номер в базовом блоке. Таким образом, если система даст сбой в ходе операций записи в куст, при следующей перезагрузке системы диспетчер конфигурации заметит, что два порядковых номера в базовом блоке куста не соответствуют друг другу. Диспетчер конфигурации может обновить куст изменившимися секторами в файле регистрации куста, чтобы прокрутить куст вперед. Тогда куст станет соответствующим текущему положению вещей.

В загрузчике Windows Boot Loader также есть код, относящийся к обеспечению достоверности реестра. Например, он может провести разбор файла System.log до загрузки ядра и провести восстановительные действия, чтобы вернуть реестру соответствие. Кроме того, в некоторых случаях повреждения куста (например, если базовый блок, приемник или ячейка содержат данные, не прошедшие проверок соответствия) диспетчер конфигурации может заново инициализировать поврежденные структуры данных, возможно, удаляя при этом подразделы, и продолжит нормальную работу. Если приходится прибегать к операции самовосстановления, диспетчер конфигурации выводит диалоговое окно ошибки, оповещая об этом пользователя.

Фильтрация реестра

Диспетчер конфигурации в ядре Windows реализует эффективную модель фильтрации реестра, позволяющую отслеживать активность реестра такими инструментами, как Process Monitor. Когда драйвер использует механизм обратного вызова, он с помощью диспетчера конфигурации регистрирует функцию обратного вызова. Диспетчер конфигурации выполняет принадлежащую драйверу функцию обратного вызова до и после работы системных служб реестра, чтобы драйвер был полностью виден и управляем через обращения к реестру. Еще одним примером использования механизма обратных вызовов могут послужить антивирусные программы, сканирующие данные реестра на отсутствие вирусов или предотвращающие изменение реестра со стороны неавторизованных процессов.

Функции обратного вызова реестра также связаны с понятием высот. Высоты являются способом для различных поставщиков зарегистрировать «высоту» в стеке фильтрации реестра, чтобы порядок, в котором происходят системные вызовы каждой процедуры обратного вызова, мог быть обусловленным и правильным. Это исключает такое развитие событий, при котором антивирусная программа сканировала бы зашифрованные разделы, прежде чем программа шифрования запустит свою собственную функцию обратного вызова для их расшифровки. При использовании модели обратных вызовов реестра Windows оба типа инструментальных средств назначают базовый уровень высоты, соответствующий типу осуществляемой ими фильтрации — в данном случае расшифровки по сравнению со сканированием. Что так же немаловажно, компании, создающие инструментальные средства такого рода, должны регистрироваться в компании Microsoft, чтобы они не вступали в конфликт с аналогичными или конкурирующими продуктами.

Модель фильтрации также включает возможность либо полностью перенять выполнение операции над реестром (в обход диспетчера конфигурации, не давая ему обрабатывать запрос), либо перенаправить операцию на другую операцию (например, при перенаправлении реестра, выполняемого Wow64). Кроме того, можно изменить выходные параметры, а также значение, возвращаемое операцией над реестром.

И наконец, драйверы могут в своих интересах назначить или пометить определяемую для драйвера информацию того или иного раздела или для той или иной операции. Драйвер может создать и назначить данные контекста в ходе операции создания (create) или открытия (open), которые будут запомнены диспетчером конфигурации и возвращены в ходе каждой последующей операции над разделом.

Оптимизации реестра

Диспетчер конфигурации проводит несколько заслуживающих внимание оптимизаций производительности. Начнем с того, что, в сущности, каждый раздел реестра имеет дескриптор безопасности, который защищает доступ к разделу. Но сохранение уникальной копии дескриптора безопасности для каждого раздела в кусте будет слишком неэффективным, поскольку одни и те же настройки безопасности зачастую применяются к целым поддеревьям реестра. Когда система обеспечивает безопасность раздела, диспетчер конфигурации проверяет

пул уникальных дескрипторов безопасности, используемых в том же кусте, к которому относится тот раздел, к которому будет применен новый дескриптор безопасности. Далее диспетчер совместно использует для этого раздела какой-нибудь уже существующий дескриптор, обеспечивая наличие в кусте не более одной копии каждого уникального дескриптора безопасности.

Диспетчер конфигурации также оптимизирует способ хранения раздела и имен параметров в кусте. Хотя реестр полноценно использует Unicode и определяет все имена, используя Unicode-соглашение, если имя состоит только из ASCII-символов, диспетчер конфигурации сохраняет имя в кусте в форме ASCI. Когда диспетчер конфигурации считывает имя (например, при поиске по имени), он преобразует в памяти это имя в форму Unicode. Хранение имени в форме ASCII может существенно сократить размер куста.

Для минимизации объема используемой памяти блоки управления разделами не сохраняют полные путевые имена разделов реестра. Вместо этого они ссылаются только на имя раздела. Например, блок управления раздела, который ссылается на \Registry\System\Control, будет ссылаться только на имя Control, а не на все путевое имя. Дальнейшая оптимизация использования памяти заключается в том, что диспетчер конфигурации использует для хранения имен разделов блоки управления именами разделов, и все блоки управления разделами для разделов с одинаковыми именами совместно используют одни и те же блоки управления именами разделов. Для оптимизации производительности диспетчер конфигурации сохраняет имена блоков управления разделами в хэш-таблице с целью ускорения поисковых операций.

Для обеспечения быстрого доступа к блокам управления разделами диспетчер конфигурации сохраняет часто востребуемые блоки управления разделами в таблице в кэше, который сконфигурирован в виде хэш-таблицы. Когда диспетчеру конфигурации нужно найти блок управления разделом, он сначала проверяет таблицу в кэше. И наконец, у диспетчера конфигурации есть еще один кэш с таблицей отложенных закрытий, в которой хранятся блоки управления разделами, закрытые приложениями, чтобы приложение могло быстро снова открыть недавно закрытый раздел. Для оптимизации поисковых операций эти таблицы в кэше хранятся для каждого куста. Диспетчер конфигурации удаляет самые старые блоки управления разделами из таблицы отложенных закрытий по мере добавления в эту таблицу недавно закрытых блоков.

Службы

Почти в каждой операционной системе есть механизм, запускающий при старте самой системы те процессы, которые предоставляют службы, не связанные с интерактивным пользователем. В Windows такие процессы называются службами или Windows-службами, потому что они для взаимодействия с системой используют API-функции Windows. Службы похожи на имеющиеся в UNIX процессы-демоны и часто реализуются на серверной стороне клиент-серверных приложений. В качестве примера Windows-службы можно назвать веб-сервер, потому что он должен быть запущен независимо от того, зарегистрировался ли кто-нибудь на компьютере или нет, и должен быть запущен при старте системы, чтобы администратору не нужно было даже помнить о необходимости его запуска или запускать его самостоятельно.

Windows-службы состоят из трех компонентов: приложения службы, программы управления службой — service control program (SCP) — и диспетчера управления службами — service control manager (SCM). Сначала будут рассмотрены приложения служб, учетные записи служб и операции SCM. Затем будет рассмотрен автозапуск служб в ходе загрузки системы. Также будут рассмотрены действия, предпринимаемые SCM при сбое службы в ходе ее запуска, и способ, используемый SCM для остановки служб.

Приложения служб

Приложения служб, например веб-серверы, состоят как минимум из одного исполняемого файла, запускаемого в качестве Windows-службы. Пользователь, желающий запустить, остановить или настроить службу, использует SCP. Хотя Windows поставляет встроенные SCP, обеспечивающие запуск, остановку, перевод в режим паузы и в режим продолжения функционирования, некоторые приложения служб включают свою собственную SCP, позволяющую администраторам определять конфигурационные настройки конкретно той службы, которой они управляют.

Приложения служб являются простыми исполняемыми файлами Windows (имеющими графический интерфейс пользователя или запускаемые из консоли) с дополнительным кодом для получения команд от SCM, а также возвращения SCM состояния службы. Поскольку у большинства служб отсутствует интерфейс пользователя, они созданы в виде консольных программ.

При установке приложения, включающего в себя службу, программа установки должна зарегистрировать службу в системе. Для регистрации службы программа установки вызывает Windows-функцию CreateService, связанную со службами функцию, которая находится в библиотеке Advapi32.dll (%SystemRoot%\System32\Advapi32.dll). В Advapi32 (DLL-библиотеке улучшенных API-функций — «Advanced API») реализованы все API-функции SCM, выполняемые на стороне клиента.

Когда программа установки регистрирует службу, вызывая функцию CreateService, SCM на той машине, где будет располагаться служба, отправляется сообщение. После этого SCM создает для службы раздел реестра в разделе HKLM\SYSTEM\CurrentControlSet\Services. Раздел Services является долговременным представлением базы данных SCM. Отдельные разделы для каждой службы определяют путь к исполняемому образу, в котором содержится служба, а также параметры и настройки конфигурации.

После создания службы установочное или управляющее приложение может запустить службу с помощью функции StartService. Поскольку некоторые приложения, основанные на работе служб, также должны в процессе загрузки системы инициализировать свою работу, нет ничего необычного в том, что программа установки регистрирует службу в качестве автозапускаемой, запрашивая у пользователя перезагрузку системы для завершения установки службы и разрешения SCM запустить службу в ходе загрузки системы.

Когда программа вызывает функцию CreateService, она должна указать ряд параметров, описывающих характеристики службы. Эти характеристики включают в себя:

- тип службы (служба относится к запускаемым в собственном процессе, а не к тем, которые делят процесс с другими службами);

- размещение файла исполняемого образа службы;
- необязательное отображаемое имя службы и пароль, используемый для запуска службы в конкретном контексте безопасности учетной записи;
- тип запуска, который показывает, должна ли служба автоматически запускаться в ходе загрузки системы или она должна запускаться произвольно по указанию SCP;
- код ошибки, показывающий, как система должна реагировать при обнаружении службой ошибки в процессе запуска;
- и, если служба запускается автоматически, дополнительную информацию, указывающую, когда служба должна запускаться относительно других служб.

SCM сохраняет каждую характеристику в виде параметра в разделе реестра, относящемся к службе. На рис. 4.5 показывается пример раздела реестра, относящегося к службе.

В табл. 4.7 перечислены все характеристики служб, многие (но не все) из которых применяются также и к драйверам устройств. Если служба нуждается в хранении конфигурационной информации, принадлежащей исключительно этой службе, то по соглашению должен создаваться подраздел Parameters, принадлежащий разделу этой службы, а затем в параметрах этого подраздела сохраняется конфигурационная информация. После этого служба может извлекать параметры, используя для этого стандартные функции реестра.

Рис. 4.5. Пример раздела реестра, относящегося к службе

ПРИМЕЧАНИЕ

SCM не обращается к подразделу Parameters той или иной службы, пока служба не будет удалена. Тогда SCM удаляет весь раздел, относящийся к службе, включая такие подразделы, как Parameters.

Таблица 4.7. Параметры реестра, относящиеся к службам и драйверам

Параметр	Значение	Описание параметра
Start	SERVICE_BOOT_START (0)	Winload заранее загружает драйвер, чтобы он был в памяти в ходе загрузки системы. Такие драйверы инициализируются непосредственно перед драйверами SERVICE_ SYSTEM_ START
	SERVICE_SYSTEM_ START (1)	Драйвер загружается и инициализируется в ходе инициализации ядра после инициализации драйверов SERVICE_ BOOT_START
	SERVICE_AUTO_START (2)	SCM запускает драйвер или службу после запуска SCM-процесса Services.exe
	SERVICE_DEMAND_ START (3)	SCM запускает драйвер или службу по требованию
	SERVICE_DISABLED (4)	Драйвер или служба не загружается или не инициализируется
ErrorControl	SERVICE_ERROR_ IGNORE (0)	Любые возвращаемые драйвером или службой ошибки игнорируются, и не регистрируются, и не отображаются никакие предупреждения
	SERVICE_ERROR_ NORMAL (1)	Если драйвер или служба сообщает об ошибке, записывается сообщение в журнал событий
	SERVICE_ERROR_ SEVERE (2)	Если драйвер или служба возвращает ошибку и последние удачные настройки конфигурации не использовались, перезагрузить с последними удачными настройками конфигурации, в противном случае продолжить загрузку системы
	SERVICE_ERROR_ CRITICAL (3)	Если драйвер или служба возвращает ошибку и последние удачные настройки конфигурации не использовались, перезагрузить с последними удачными настройками конфигурации, в противном случае произойдет остановка загрузки системы с отображением синего аварийного экрана
Type	SERVICE_KERNEL_ DRIVER (1)	Драйвер устройства
	SERVICE_FILE_ SYSTEM_DRIVER (2)	Файл системного драйвера режима ядра
	SERVICE_ADAPTER (4)	Устаревший параметр
	SERVICE_ RECOGNIZER_DRIVER (8)	Драйвер распознавания файловой системы

продолжение ⇩

Таблица 4.7 *(продолжение)*

Параметр	Значение	Описание параметра
	SERVICE_WIN32_ OWN_PROCESS (16)	Служба запускается в своем собственном индивидуальном процессе
	SERVICE_WIN32_ SHARE_PROCESS (32)	Служба запускается в процессе, общем для нескольких служб
	SERVICE_ INTERACTIVE_ PROCESS (256)	Службе разрешается выводить окно консоли и получать пользовательский ввод, но это касается только консоли сеанса (0) для предотвращения взаимодействия с пользовательскими или консольными приложениями в других сеансах
Group	Имя группы	Драйвер или служба инициализируются, когда инициализируется их группа
Tag	Номер тега	Указанное место в порядке инициализации группы. Этот параметр к службам не применяется
ImagePath	Пути к исполняемому файлу службы или драйвера	Если параметр ImagePath не определен, диспетчер ввода-вывода ищет драйверы в %SystemRoot%\System32\Drivers. Требуется для служб Windows
Depend-OnGroup	Имя группы	Драйвер или служба не будет загружаться, пока не загрузится драйвер или служба из указанной группы
Depend-OnService	Имя службы	Служба не будет загружаться, пока не будет загружена указанная служба. Этот параметр не применяется к драйверам устройств, кроме тех, чей тип запуска не имеет значение SERVICE_ AUTO_START или SERVICE_DEMAND_ START
ObjectName	Обычно имеет значение LocalSystem, но это может быть имя учетной записи, например .\Administrator	Определяет учетную запись, под которой будет работать служба. Если параметр ObjectName не определен, используется учетная запись LocalSystem. Этот параметр к драйверам устройств не применяется
DisplayName	Имя службы	Под этим именем служба показывается своим приложением. Если имя не определено, в качестве имени используется имя раздела реестра
Description	Описание службы	Описание службы длиной до 32 767 байт
FailureActions	Описание действий, которые должны выполняться SCM, когда выход из процесса службы происходит неожиданно	К числу действий в случае сбоя относятся перезапуск процесса службы, перезагрузка системы и запуск указанной программы. К драйверам этот параметр не применяется

Параметр	Значение	Описание параметра
FailureCommand	Командная строка программы	SCM считывает этот параметр, только если указан параметр FailureActions, указания которого программа должна выполнить в случае сбоя службы. К драйверам этот параметр не применяется
DelayedAutoStart	0 или 1 (TRUE или FALSE)	Предписывает SCM запуск этой службы после некоторой задержки от времени запуска SCM. Тем самым сокращается количество служб, запускаемых одновременно с запуском SCM
PreshutdownTimeout	Лимит времени в миллисекундах	Этот параметр позволяет службам заменить используемый по умолчанию лимит времени в 180 секунд на уведомление перед остановкой. После истечения данного лимита времени SCM остановит работу службы, если она так и не откликнется
ServiceSidType	SERVICE_SID_TYPE_NONE (0)	Настройка, предназначенная для обратной совместимости
	SERVICE_SID_TYPE_UNRESTRICTED (1)	При создании службы SCM ее SID группового владельца к маркеру служебного процесса
	SERVICE_SID_TYPE_RESTRICTED (3)	То же самое, что и выше, но SCM также добавляет SID службы к ограниченному списку SID служебного процесса, наряду с SID всех пользователей (world), входа в систему (logon) и ограничений по записи (write-restricted)
RequiredPrivileges	Список привилегий	Это параметр содержит список привилегий, которые требуются службе для ее функционирования. SCM будет вычислять их слияние при создании маркера для общего процесса, относящегося к этой службе, если таковой имеется
Security	Дескриптор безопасности	Этот параметр содержит необязательный дескриптор безопасности, который определяет, у кого какие имеются права доступа к объекту службы, созданному внутри системы диспетчером SCM. Если этот параметр опущен, SCM применяет дескриптор безопасности, используемый по умолчанию

Обратите внимание на то, что параметр Type включает три значения, применимые к драйверам устройств: драйвер устройства, драйвер файловой системы и распознаватель файловой системы. Они используются драйверами устройств Windows, которые также сохраняют свои параметры в виде данных реестра в разделе реестра Services. SCM отвечает за запуск драйверов с помощью значений параметра Start SERVICE_AUTO_START или SERVICE_DEMAND_START, поэтому нет ничего необычного в том, что база данных SCM включает в себя драйверы. Службы используют другие типы, SERVICE_WIN32_OWN_PROCESS и SERVICE_WIN32_SHARE_PROCESS, которые являются взаимоисключающими. Исполняемые файлы, вмещающие более одной службы, относятся к типу SERVICE_WIN32_SHARE_PROCESS.

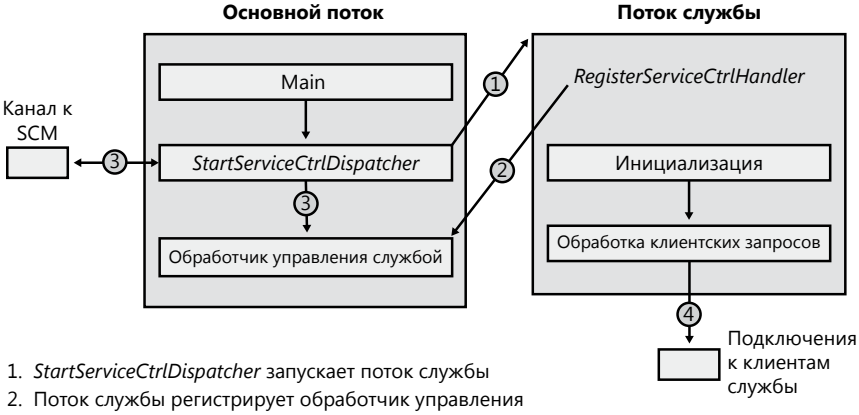
Преимущество от запуска в одном процессе более одной службы состоит в экономии тех системных ресурсов, которые потребовались бы для их запуска в отдельных процессах. Потенциальный недостаток состоит в том, что когда в одной из служб коллекции, запущенной в одном процессе, возникает ошибка, она становится причиной завершения всего процесса. Еще одно ограничение заключается в том, что все службы должны работать под одной и той же учетной записью (но, если служба пользуется механизмами ужесточения безопасности служб, это может ограничить некоторые из ее уязвимостей от вредоносных атак).

Когда SCM запускает процесс службы, этот процесс должен сразу же вызвать функцию StartServiceCtrlDispatcher. Эта функция получает список точек входа в службы, по одной точке входа для каждой службы в процессе. Каждая точка входа идентифицируется с помощью имени службы, к которой она относится. После создания именного канала связи с SCM StartServiceCtrlDispatcher ожидает команд, поступающих по каналу от SCM. Диспетчер SCM отправляет команду на запуск службы при каждом запуске службы, которой владеет процесс. При получении каждой команды запуска функция StartServiceCtrlDispatcher создает поток, называемый служебным потоком, чтобы передать управление в стартовую точку службы и реализовать для этой службы командный цикл. Функция StartServiceCtrlDispatcher переходит к бесконечному ожиданию команд от SCM и возвращает управление основной функции процесса только когда останавливаются все службы процесса, позволяя служебному процессу освободить ресурсы перед выходом.

В первую очередь из точки входа вызывается функция RegisterServiceCtrlHandler. Эта функция получает и сохраняет указатель на функцию, которую служба реализует для обработки различных команд, получаемых от SCM. Эта функция называется обработчиком управления. Обработчик RegisterServiceCtrlHandler не обменивается данными с SCM, а сохраняет функцию в памяти локального процесса для функции StartServiceCtrlDispatcher. Код точки входа в службу продолжает инициализацию службы, что может включать в себя выделение памяти, создание конечных точек обмена данными и чтение индивидуальных конфигурационных данных из реестра. Как уже ранее говорилось, по соглашению, которого придерживается большинство служб, их параметры хранятся в подразделе Parameters, который находится в разделе реестра службы.

Когда код точки входа инициализирует службу, он должен периодически отправлять сообщения SCM о состоянии, используя функцию SetServiceStatus, показывающие прогресс в запуске службы. После того как код точки входа завершит инициализацию, поток службы обычно находится в цикле ожидания запросов от клиентских приложений. Например, веб-сервер инициализирует сокет прослушивания TCP и ждет поступающих запросов на HTTP-подключения.

Основной поток служебного процесса, в котором выполняется функция StartServiceCtrlDispatcher, получает команды SCM, направляемые службам в процессе, и вызывает функцию обработки управления нужной службы, сохраненную RegisterServiceCtrlHandler. SCM-команды включают в себя команду остановки (stop), перевода в режим паузы (pause), возобновления работы (resume), опроса (interrogate) и завершения работы (shutdown) или команды, определенные приложением. На рис. 4.6 показана внутренняя организация процесса службы. На нем изображены два потока, из которых состоит процесс, в котором выполняется одна служба: основной поток и поток службы.



1. *StartServiceCtrlDispatcher* запускает поток службы
2. Поток службы регистрирует обработчик управления
3. *StartServiceCtrlDispatcher* вызывает обработчики в ответ на команды SCM
4. Поток службы обрабатывает клиентские запросы

Рис. 4.6. Внутренняя организация процесса службы

Учетные записи служб

Контекст безопасности службы играет важную роль как для разработчиков служб, так и для системных администраторов, потому что он определяет, к каким ресурсам процесс сможет получить доступ. Пока программа установки службы или администратор не укажут что-нибудь другое, большинство служб запускаются в контексте безопасности учетной записи локальной системы (иногда отображается как SYSTEM, а иногда как LocalSystem). Две другие встроенные учетные записи относятся к сетевой службе и локальной службе. С точки зрения безопасности у этих учетных записей меньше возможностей, чем у учетной записи локальной системы. Любая встроенная служба Windows, не требующая всех прав, которые имеются у учетной записи локальной системы, запускается под соответствующей альтернативной учетной записью службы. В следующих подразделах рассматриваются особенности этих учетных записей.

Учетная запись локальной системы

Учетная запись локальной системы является той самой учетной записью, под которой запускаются основные компоненты операционной системы Windows, работающие в пользовательском режиме, включая диспетчер сеансов (%SystemRoot%\System32\Smss.exe), процесс подсистемы Windows (Csrss.exe), процесс авторизации локальных пользователей — Local Security Authority process (%SystemRoot%\System32\Lsass.exe) — и процесс входа в систему — Logon process (%SystemRoot%\System32\Winlogon.exe). Дополнительные сведения о последних двух процессах даны в главе 6.

С точки зрения безопасности, когда речь идет о возможностях обеспечения безопасности локальной системы, учетная запись локальной системы особенно эффективна. Она намного эффективнее, чем любая локальная или доменная учетная запись. Эта учетная запись имеет следующие характеристики:

- Она входит в группу локальных администраторов. В табл. 4.8 показаны группы, в которые входит учетная запись локальной системы.

- Ее обладатель имеет право задействовать практически каждую привилегию (даже те привилегии, которые обычно не даются обладателю учетной записи локального администратора, например, по созданию маркеров доступа). Список привилегий, выделяемых обладателю учетной записи локальной системы показан в табл. 4.9.

Таблица 4.8. Групповое членство учетных записей, под которыми запускаются службы

Локальная система (Local System)	Сетевая служба (Network Service)	Локальная служба (Local Service)
Everyone	Everyone	Everyone
Authenticated Users	Authenticated Users	Authenticated Users
Administrators	Users	Users
	Local	Local
	Local Service	Local Service
	Service	Service

Таблица 4.9. Привилегии, получаемые при работе под учетной записью службы

Локальная система (Local System)	Сетевая служба (Network Service)	Локальная служба (Local Service)
SeAssignPrimaryTokenPrivilege SeAuditPrivilege SeBackupPrivilege SeChangeNotifyPrivilege SeCreateGlobalPrivilege SeCreatePagefilePrivilege SeCreatePermanentPrivilege SeCreateTokenPrivilege SeDebugPrivilege SeImpersonatePrivilege SeIncreaseBasePriorityPrivilege SeIncreaseQuotaPrivilege SeLoadDriverPrivilege SeLockMemoryPrivilege SeManageVolumePrivilege SeProfileSingleProcessPrivilege SeRestorePrivilege SeSecurityPrivilege SeShutdownPrivilege SeSystemEnvironmentPrivilege SeSystemTimePrivilege SeTakeOwnershipPrivilege SeTcbPrivilege SeUndockPrivilege (только для клиентов)	SeAssignPrimaryToken-Privilege SeAuditPrivilege SeChangeNotifyPrivilege SeCreateGlobalPrivilege SeImpersonatePrivilege SeIncreaseQuotaPrivilege SeShutdownPrivilege SeUndockPrivilege (только для клиентов) Привилегии, выделяемые группам Everyone, Authenticated Users и Users	SeAssignPrimaryToken-Privilege SeAuditPrivilege SeChangeNotifyPrivilege SeCreateGlobalPrivilege SeImpersonatePrivilege SeIncreaseQuotaPrivilege SeShutdownPrivilege SeUndockPrivilege (только для клиентов) Привилегии, выделяемые группам Everyone, Authenticated Users и Users

- У обладателя учетной записи локальной системы имеется полный доступ к большинству файлов и разделов реестра. Даже если он не получает полный доступ, процесс, запущенный под учетной записью локальной системы, может воспользоваться привилегией получения монопольного доступа.
- Процессы, запущенные под учетной записью локальной системы, запускаются с исходным профилем пользователя (HKU\.DEFAULT). Поэтому они не могут обращаться к конфигурационной информации, хранящейся в профилях пользователей других учетных записей.
- Когда система является участником домена Windows, учетная запись локальной системы включает в себя идентификатор безопасности машины (SID) того компьютера, на котором запущен процесс службы. Поэтому служба, запущенная под учетной записью локальной системы, получит автоматическую аутентификацию на других машинах в той же группе доменов (в том же лесу) за счет использования свой учетной записи компьютера.
- Если учетной записи компьютера специально предоставлен доступ к ресурсам (например, к сетевым ресурсам, именованным каналам и т. д.), процесс может обращаться к сетевым ресурсам, которые допустимы из нулевых сеансов, то есть подключений, не требующих никаких полномочий. Указать общие ресурсы и каналы на конкретном компьютере, разрешенные нулевым сеансам, можно в параметрах реестра NullSessionPipes и NullSessionShares, которые находятся в разделе HKLM\SYSTEM\CurrentControlSet\Services\lanmanserver\parameters.

Учетная запись сетевой службы (Network Service)

Учетная запись сетевой службы предназначена для тех служб, которым нужно пройти аутентификацию на других машинах сети, используя учетную запись компьютера, так же как и учетную запись локальной системы, но при этом не требуется членство в группе администраторов или не нужно пользоваться множеством привилегий, которыми наделяется владелец учетной записи локальной системы. Поскольку учетная запись сетевой службы не входит в группу администраторов, службы, запущенные под учетной записью сетевой службы, исходно имеют доступ к гораздо меньшему количеству разделов реестра или папок файловой системы и файлов, чем службы, запущенные под учетной записью локальной системы. Кроме того, наделение незначительным числом привилегий ограничивает сферу опасности процессов сетевых служб. Например, процесс, запущенный под учетной записью сетевой службы, не может загрузить драйвер устройства или открыть произвольные процессы.

Другим отличием учетной записи сетевой службы от учетной записи локальной системы является то, что процессы, запущенные под учетной записью сетевой службы, используют профиль этой учетной записи. Компонент реестра, относящийся к профилю сетевой службы, загружается в раздел HKU\S-1-5-20, и файлы и каталоги, составляющие компонент, находятся в подразделе %SystemRoot%\ServiceProfiles\NetworkService.

Служба, запущенная под учетной записью сетевой службы, является DNS-клиентом, отвечающим за разрешение DNS-имен и за определение местоположения контроллеров доменов.

Учетная запись локальной службы

Учетная запись локальной службы практически похожа на учетную запись сетевой службы, но есть одно важное отличие, состоящее в том, что под ней можно получить доступ только к тем сетевым ресурсам, которые допускают анонимный доступ. В табл. 4.9 показано, что учетная запись сетевой службы дает те же привилегии, что и учетная запись локальной службы, а в табл. 4.8 показано, что она принадлежит к тем же группам, за исключением того, что она принадлежит не к группе Local Service, а к группе Network Service. Профиль, используемый процессами, запущенными под учетной записью локальной службы, загружается в раздел HKU\S-1-5-19 и хранится в подразделе %SystemRoot%\ServiceProfiles\LocalService.

Примеры служб, запускаемых под учетной записью, включают в себя службу удаленного реестра — Remote Registry Service, допускающую удаленный доступ к реестру локальной системы, и службу LmHosts, осуществляющую разрешение имен NetBIOS.

Запуск служб под другими учетными записями

В силу уже упомянутых ограничений, некоторым службам необходим запуск с полномочиями безопасности учетной записи пользователя. Можно настроить службу на запуск под другой учетной записью при ее создании или путем указания с помощью Windows-оснастки Консоли управления Службы (Windows Services MMC) учетной записи и пароля, под которыми служба должна работать. В оснастке Службы нужно щелкнуть правой кнопкой мыши на службе, выбрать пункт Свойства (Properties), щелкнуть на вкладке Вход в систему (Log On) и выбрать пункт С учетной записью (This Account), как показано на рис. 4.7.

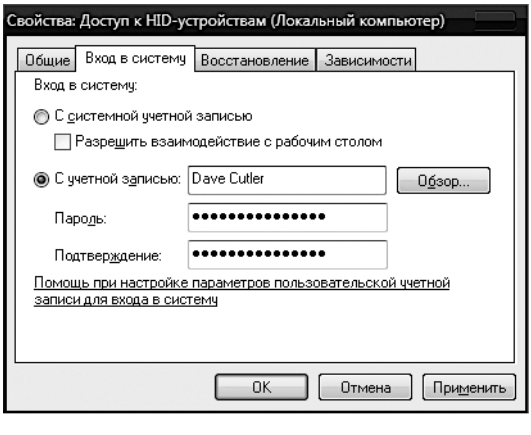


Рис. 4.7. Настройки учетной записи, под которой будет запущена служба

Запуск с наименьшими привилегиями

Службы обычно подчиняются модели «все или ничего», означающей, что все привилегии, доступные той учетной записи, под которой запущен процесс службы, доступны службе, запущенной в процессе, которой может понадобиться только лишь какая-то часть этих привилегий. Чтобы лучше соответствовать принципу

наименьших привилегий, согласно которому Windows назначает службам только нужные им привилегии, разработчики могут указать только те привилегии, которые требуются их службе, и SCM создает маркер доступа, который содержит только эти привилегии.

ПРИМЕЧАНИЕ

Привилегии, указанные для службы, должны быть поднабором тех привилегий, которые доступны той учетной записи, под которой эта служба запускается.

Разработчики служб используют для указания списка требуемых привилегий API-функцию ChangeServiceConfig2. API-функция сохраняет эту информацию в реестре в разделе Parameters, созданном для этой службы. При запуске службы SCM считывает раздел и добавляет привилегии к маркеру того процесса, в котором эта служба запускается.

Если существует параметр RequiredPrivileges и служба относится к автономным (запускаемым в виде отдельного процесса), SCM создает маркер, содержащий только те привилегии, которые ей нужны. Для служб, запускаемых в процессе, служащем для запуска сразу нескольких служб, для которых указаны требуемые привилегии, SCM вычисляет сумму этих привилегий и объединяет их для маркера доступа того процесса, в котором запущены службы. Иными словами, будут удалены только те привилегии, которые не указаны ни для одной их служб, являющихся частью группы служб. Если параметр реестра не существует, SCM может лишь посчитать, что служба несовместима с моделью предоставления наименьших привилегий или же требует для своей работы всех привилегий. В таком случае создается маркер полного доступа, содержащий все привилегии, и никаких дополнительных мер безопасности этой моделью не предлагается. Чтобы убрать почти все привилегии, службы могут указать лишь привилегию уведомлений об изменениях — Change Notify.

ЭКСПЕРИМЕНТ: ПРОСМОТР ПРИВИЛЕГИЙ, ТРЕБУЕМЫХ СЛУЖБАМ

Просмотреть требуемые службам привилегии можно с помощью утилиты управления службами — Service Control, Sc.exe и ключа qprivs. Кроме этого, информацию о маркере доступа любого, имеющегося в системе процесса службы, может показать Process Explorer, позволяя вам сравнить информацию, возвращенную Sc.exe с частью привилегий маркера доступа. Для некоторых наиболее подходящих заблокированных служб системы это можно сделать, выполнив следующие действия:

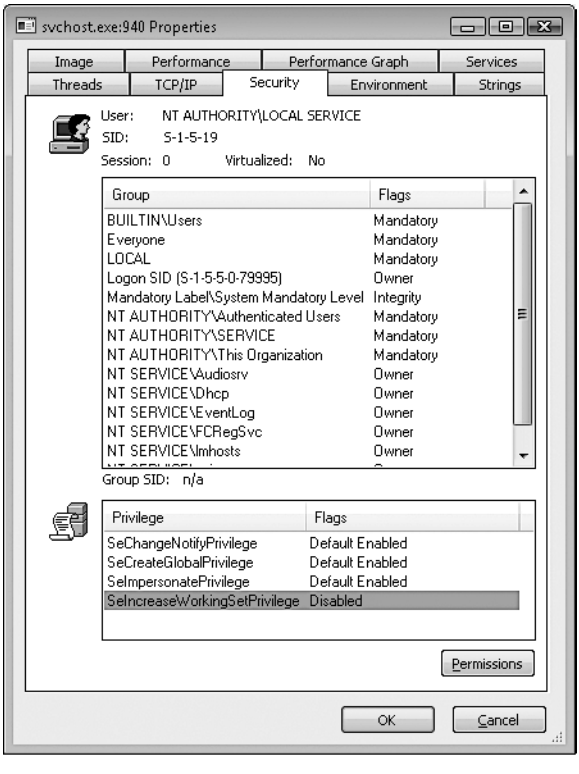
1. Для просмотра требуемых привилегий, указанных для службы Dhcp, воспользуйтесь утилитой Sc.exe, для чего введите в командную строку следующую команду:

```
sc qprivs dhcp
```

   Должны появиться две запрошенные привилегии: SeCreateGlobalPrivilege и SeChangeNotifyPrivilege.

2. Запустите Process Explorer и посмотрите на список процессов. Должны показаться два процесса Svchost.exe, в которых выполняются службы на вашей машине. Process Explorer выделит их розовым цветом.

3. Теперь найдите процесс, в котором запущена служба Dhcp. Она должна быть запущена неподалеку от других служб, являющихся частью группы служб LocalServiceNetworkRestricted, таких как служба Audiosrv и служба Eventlog. Сделать это можно, наводя указатель мыши на каждый Svchost процесс и читая появляющуюся подсказку, в которой содержатся имена служб, запущенных внутри хост-процесса (процесса-хозяина).
4. Как только процесс будет найден, дважды щелкните на его имени, чтобы открыть диалоговое окно Properties (Свойства) и выбрать вкладку Security (Безопасность).



Заметьте, что хотя служба запущена в составе служб, работающих под учетной записью локальной службы, перечень привилегий, которые назначены ей со стороны Windows, намного короче перечня привилегий, доступных учетной записи локальных служб, показанного в табл. 4.9.

Поскольку та часть, которая относится в маркере доступа хост-процесса служб к привилегиям, является объединением привилегий, требуемых всем запущенным в нем службам, это должно означать, что такие службы как Audiosrv и Eventlog, не запрашивают привилегий, отличных от тех, которые показываются утилитой Process Explorer. Вы можете проверить это, запустив утилиту Sc.exe и в отношении этих служб. ■

Изоляция служб

Ограничение количества привилегий, к которым служба имеет доступ, помогает уменьшить негативное влияние подвергнутых риску служебных процессов на

другие процессы. Но это ничего не дает для изоляции службы от ресурсов, к которым в обычных условиях имеется доступ из той учетной записи, под которой она запущена. Как уже ранее упоминалось, у учетной записи локальной системы имеется полный доступ к важным системным файлам, разделам реестра и другим защищаемым объектам системы, поскольку списки управления доступом — access control lists (ACL) дают на это разрешение данной учетной записи.

Время от времени доступ к некоторым из этих ресурсов действительно имеет важное значение для операций, проводимых службой, но в то же время другие объекты должны быть от этой службы защищены. Ранее, чтобы не допустить работы под учетной записью локальной системы для получения доступа к необходимым ресурсам, служба должна была запускаться под стандартной учетной записью пользователя, и ACL-списки должны были добавляться к системным объектам, что существенно повышало риск атаки системы со стороны вредоносного кода. Еще одним решением было создание специально назначенных учетных записей служб и установка отдельных ACL-списков для каждой учетной записи (связанной со службой), но этот подход быстро стал приносить администраторам большие хлопоты.

Сейчас Windows сочетает два подхода в гораздо более управляемом решении: она позволяет службам запускаться под непривилегированной учетной записью, но все же иметь доступ к отдельным привилегированным ресурсам без снижения уровня безопасности подобных объектов. Как и в решении из второй предварительной версии Windows Vista, ACL-списки объекта теперь могут устанавливать разрешения непосредственно службе, а не путем запроса отдельной учетной записи. Вместо этого SCM генерирует SID, представляющий ту или иную службу, и этот SID может использоваться для установки разрешений на доступ к таким ресурсам, как разделы реестра и файлы. SID-идентификаторы службы реализованы в виде той части маркеров доступа для любых хост-процессов служб, которая относится к группе SID-идентификаторов. Они генерируются SCM в ходе запуска системы для каждой службы, для которой требуется идентификатор безопасности, для чего используется API-функция ChangeServiceConfig2. Когда речь идет о хост-процессах (содержащих более одной службы), маркер доступа процесса будет содержать SID-идентификаторы всех служб, являющихся частью группы служб, связанных с процессом, включая те службы, которые еще не запущены, поскольку способа добавления новых SID после создания маркера не существует.

Польза от наличия SID для каждой службы выходит за рамки простой возможности добавления записей в ACL-список и разрешений для различных объектов системы как способа получения детального управления доступом. В начале рассмотрения данного вопроса затрагивался случай, когда конкретные объекты системы, доступные при работе под заданной учетной записью, должны быть защищены от службы, запущенной под той же учетной записью. Согласно описанию, SID-идентификаторы служб препятствуют возникновению данной проблемы только путем реализации следующего требования: чтобы записи отказа — Deny, — связанные с SID службы, помещались на каждом объекте, который нуждается в защите. Но этот подход является явно неуправляемым.

Чтобы в качестве средства предотвращения доступа служб к тем ресурсам, к которым имеется доступ у пользовательской учетной записи, под которой они были запущены, не приходилось использовать запрещающие элементы управления доступом — Deny access control entries (ACE), — существует два типа SID-идентификаторов служб: ограниченный SID службы (restricted service SID,

SERVICE_SID_TYPE_RESTRICTED) и неограниченный SID службы (unrestricted service SID, SERVICE_SID_TYPE_UNRESTRICTED). Последний является используемым по умолчанию, и именно его мы до сих пор и рассматривали.

Неограниченные SID служб создаются как «включенные по умолчанию» SID-идентификаторы групповой принадлежности, и маркеру процесса также дается новый ACE-элемент, предоставляющий полное разрешение SID-идентификатору службы входа в систему, что дает возможность службе продолжить обмен данными с SCM. Главным использованием этой возможности будет включение или выключение SID-идентификаторов служб внутри процесса в ходе запуска или остановки службы.

С другой стороны, ограниченные SID служб превращают маркер хост-процесса служб в маркер с ограничением записи (см. главу 6). Это означает, что только объекты, предоставляющие явное право доступа по записи SID службы, будут доступны службе по записи, независимо от той учетной записи, под которой служба была запущена. По этой причине все службы, запущенные внутри этого процесса (являющиеся частью одной и той же группы служб), должны иметь ограниченный тип SID, в противном случае службы с ограниченным типом SID не смогут стартовать. Как только маркер становится ограниченным по записи, из соображения совместимости добавляются еще три SID-идентификатора:

- SID всех пользователей (world SID) добавлен, чтобы разрешить доступ по записи к объектам, которые обычно доступны кому угодно и в любом случае, самое важное, конкретным DLL-библиотекам в пути загрузки.
- SID службы входа в систему добавлен, чтобы позволить службе обмениваться данными с SCM.
- SID, ограниченный по записи, добавлен, чтобы позволить объектам явно разрешать любым, ограниченным по записи службам иметь к ним доступ по записи. Например, трассировка событий для Windows — Event Tracing for Windows (ETW) — использует этот SID на своих объектах, чтобы разрешить любой службе, ограниченной по записи, генерировать события.

На рис. 4.8 показан пример хост-процесса служб, содержащего службы, которые были помечены как имеющие ограниченные SID-идентификаторы служб. Например, основной механизм фильтрации — Base Filtering Engine (BFE), — отвечающий за применение правил фильтрования брандмауэра Windows Firewall, является частью этой службы, потому что эти правила сохранены в разделах реестра, которые должны быть защищены от вредоносного доступа по записи, если службе будет что-нибудь угрожать. (Это могло бы позволить вредоносному коду службы отключить правила брандмауэра, касающиеся исходящего трафика, разрешая, к примеру, двунаправленный обмен данными с тем, кто атакует систему.)

Путем блокирования доступа по записи к объектам, которые иначе были бы доступны службе по записи (благодаря наследованию разрешений той учетной записи, под которой эта служба запущена), ограниченные SID службы решают и другую сторону той проблемы, которую мы изначально обозначили. Пользователям не нужно ничего делать для того, чтобы не дать службе, запущенной под привилегированной учетной записью, возможность доступа по записи к важным системным файлам, разделам реестра или другим объектам, ограничивая открытость для атак со стороны любой такой службы, которая может быть скомпрометирована.

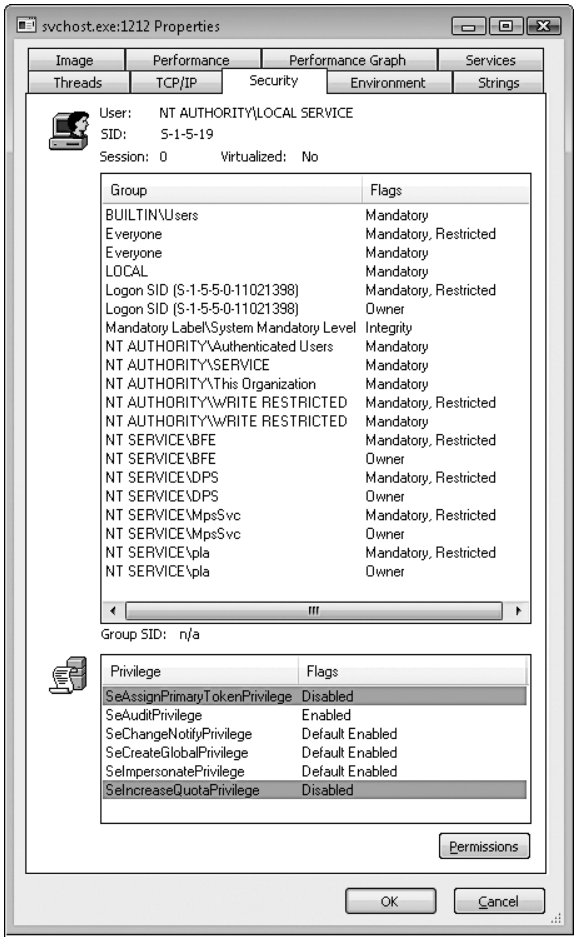


Рис. 4.8. Службы с ограниченными SID-идентификаторами служб

Windows также разрешает брандмауэру устанавливать правила, ссылающиеся на SID-идентификаторы служб, связанные с тремя типами поведения (см. табл. 4.10).

Таблица 4.10. Правила ограничений при работе в сети

Сценарий	Пример	Ограничения
Доступ к сети заблокирован	Служба обнаружения оборудования оболочки — shell hardware detection service (ShellHWDetection)	Заблокирован весь обмен данными по сети (как входящий, так и исходящий
Доступ к сети имеет статические ограничения по портам	Служба RPC (Rpcss) работает с портом 135 (TCP и UDP)	Обмен данными по сети ограничен указанными портами TCP или UDP
Доступ к сети имеет динамические ограничения по портам	Служба DNS (Dns) прослушивает изменяемые порты (UDP)	Обмен данными по сети ограничен настраиваемыми портами TCP или UDP

Интерактивные службы и изоляция нулевого сеанса (Session 0)

Одним из ограничений для служб, запущенных под учетными записями локальной системы, локальной службы и сетевой службы, является то, что эти службы не могут без использования специального флага в функции MessageBox выводить диалоговые окна на интерактивном столе пользователя. Это ограничение всегда присутствовало в Windows и не являлось прямым результатом запуска под этими учетными записями, а скорее было следствием того способа, с помощью которого подсистема Windows назначала процессы служб станциям окон (window station). Это ограничение усугубилось еще больше за счет использования сеансов в модели, названной изоляцией нулевого сеанса — Session Zero Isolation. Результатом стало то, что эти службы не могут напрямую взаимодействовать с рабочим столом пользователя.

Подсистема Windows связывает каждый Windows-процесс со станцией окна. Эта станция содержит рабочие столы, а столы содержат окна. Только одна станция окна может быть видна на консоли и может получать ввод от мыши и клавиатуры. В среде окружения служб терминалов — Terminal Services — видна только одна станция окна на сеансе, но все службы работают как часть сеанса консоли. Windows дает видимой станции окна имя WinSta0, и все интерактивные процессы обращаются к WinSta0.

Если не указано иное, подсистема Windows связывает службы, запущенные под учетной записью локальной системы, с невидимой станцией окна, которая называется Service-0x0-3e7$. Эта станция совместно используется всеми неинтерактивными службами. Число в имени, 3e7, представляет собой идентификатор сеанса входа в систему. Этот идентификатор процесс авторизации локальных пользователей — Local Security Authority process (LSASS) — назначает сеансу входа в систему, SCM использует для неинтерактивных служб, запущенных под учетной записью локальной системы.

Службы, настроенные для запуска под учетной записью пользователя (то есть не под учетной записью локальной системы), запускаются в другой невидимой станции окна. Эта станция называется LSASS по идентификатору входа в систему, назначенному сеансу службы входа в систему. На рис. 4.9 показан пример экрана утилиты Sysinternals WinObj при просмотре каталога диспетчера объектов, в который Windows помещает объекты станций окна. На этом рисунке видны интерактивная станция окна (WinSta0) и неинтерактивная станция окна системных служб (Service-0x0-3e7$).

Независимо от того, под какой учетной записью запущены службы — под пользовательской, или локальной системы, или локальной или сетевой службы, — те службы, которые не были запущены на видимой станции окна, не могут получать ввод от пользователя или отображать окна в консоли. Фактически, если служба должна была вывести обычно диалоговое окно на станцию окна, то она будет находиться в зависшем состоянии, поскольку пользователь не сможет увидеть диалоговое окно. Это, конечно же, не даст пользователю ввести что-нибудь с помощью мыши или клавиатуры для отклонения этого окна и разрешения службе продолжить выполнение.

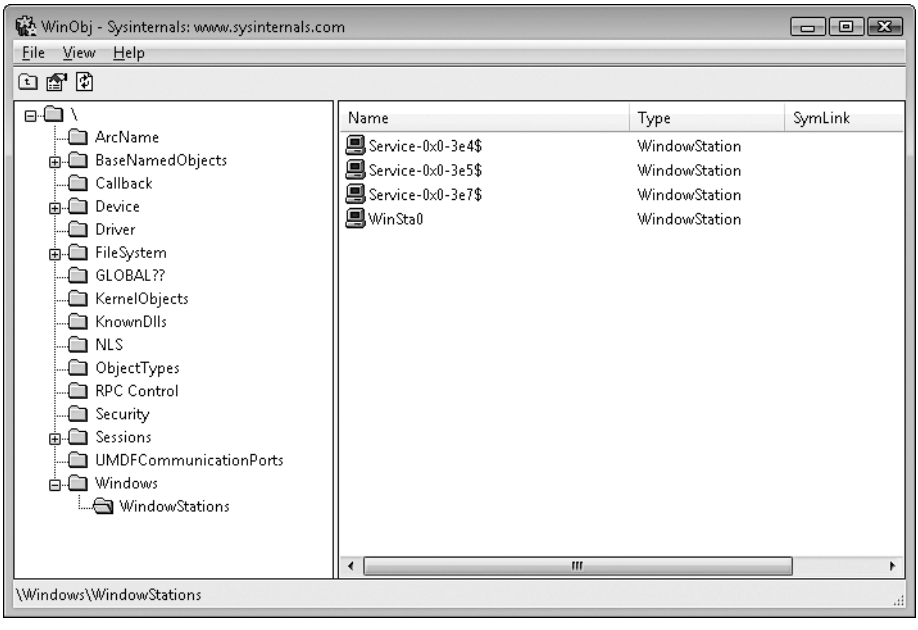


Рис. 4.9. Перечень станций окон

ПРИМЕЧАНИЕ

В прошлом была возможность воспользоваться специальными флагами MB_SERVICE_NOTIFICATION или MB_DEFAULT_DESKTOP_ONLY с помощью API-функции MessageBox для отображения сообщений на интерактивной станции окна, даже если служба была помечена неинтерактивной. Из-за изоляции сеансов любая служба, использующая этот флаг, будет тут же получать возвращаемое значение IDOK, и окно сообщения никогда не будет выведено на экран.

В редких случаях у службы может быть веская причина для взаимодействия с пользователем посредством диалоговых окон. Чтобы настроить службу на наличие права на взаимодействие с пользователем, в параметре Type, принадлежащем разделу реестра этой службы, должен присутствовать модификатор SERVICE_INTERACTIVE_PROCESS. (Учтите, что службы, настроенные на работу под пользовательской учетной записью, не могут быть помечены как интерактивные.) Когда SCM запускает службу, помеченную как интерактивная, он запускает процесс службы в контексте безопасности учетной записи локальной системы, но подключает службу к WinSta0, а не к станции окна неинтерактивной службы.

Если бы какие-то пользовательские процессы были запущены в том же самом сеансе, что и службы, это подключение к WinSta0 позволило бы службе вывести диалоговые окна на консоль и позволило бы этим окнам реагировать на пользовательский ввод, поскольку пользователи совместно использовали бы станцию окна с интерактивными службами. Но в сеансе 0 запускаются только процессы, которыми владеет система и службы Windows, а все остальные сеансы входа в систему, в том числе пользователи консоли, запускаются в других сессиях. Поэтому пользователь не видит окон, отображенных процессом в сеансе 0.

Эта дополнительная граница помогает предотвратить разрушительные атаки, посредством которых менее привилегированные приложения отправляют оконные сообщения в окно, видимое на той же станции окна. Цель подобных действий — воспользоваться дефектом в более привилегированном процессе, владеющем окном, что позволяет менее привилегированному приложению выполнить код в более привилегированном процессе.

Чтобы сохранить совместимость со службами, которые зависят от пользовательского ввода, Windows включает в себя службу, уведомляющую пользователей, когда служба отобразила окно. Служба обнаружения интерактивных служб — Interactive Services Detection (UI0Detect) — ищет видимые окна на главном рабочем столе станции окна WinSta0 сеанса 0 и выводит диалоговое окно на консоль пользовательского рабочего стола, позволяя пользователю переключиться на сеанс 0 и просматривать пользовательский интерфейс службы. Это сродни подключению к сеансу локальных служб терминалов (Terminal Services) или переключению пользователей.

ПРИМЕЧАНИЕ

Механизм обнаружения интерактивных служб создан исключительно для обеспечения совместимости приложений, и разработчикам настоятельно рекомендуется отходить от применения интерактивных служб и использовать для визуальной связи с пользователем вспомогательное, непривилегированное приложение. Для настроечных целей после получения ввода из пользовательского интерфейса между этим вспомогательным приложением и службой может использоваться локальный RPC или COM.

Диалоговое окно, пример которого показан на рис. 4.10, включает имя процесса, время, когда было выведено сообщение пользовательского интерфейса, и заголовок отображаемого окна. Как только пользователь подключается к сеансу 0, похожее диалоговое окно предоставляет возможность для возвращения в сеанс пользователя. На рисунке службой, отобразившей окно, является приложение Microsoft Paint, запущенное явным образом утилитой Sysinternals PsExec с ключами, заставляющими PsExec запустить Paint в сеансе 0. Вы можете проделать все это самостоятельно, воспользовавшись следующей командой:

```
psexec -s -i 0 -d mspaint.exe
```

Она предписывает PsExec запустить Microsoft Paint в качестве системного процесса (-s), работающего в сеансе 0 (-i 0), и сразу же вернуть управление, не ожидая завершения процесса (-d).

Если щелкнуть на пункте View The Message (Просмотреть сообщение), можно будет переключиться на консоль сеанса 0 (и переключиться назад из аналогичного окна на этой консоли).

Диспетчер управления службами

Исполнительным файлом диспетчера управления службами (SCM) является %SystemRoot%\System32\Services.exe, и, как и большинство процессов служб, он запускается как консольная программа Windows. Процесс Wininit запускает SCM на ранней стадии загрузки системы. Пусковая функция SCM, SvcCtrl-Main, управляет запуском служб, сконфигурированных на автоматический запуск.

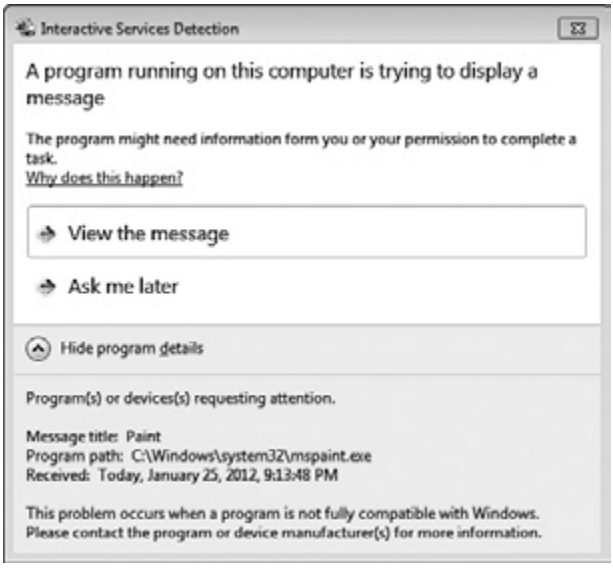


Рис. 4.10. Служба обнаружения интерактивных служб — Interactive Services Detection в работе

Функция SvcCtrlMain сначала создает событие синхронизации SvcctrlStartEvent_A3752DX, которое инициализируется в несигнальном состоянии. Только после того как SCM завершит действия, необходимые для его подготовки для получения команд от SCP-программ, он переводит событие в сигнальное состояние. Для установки диалога с SCM SCP использует функцию OpenSCManager. Эта функция предохраняет SCP от попыток контактов с SCM до того, как SCM пройдет инициализацию, путем ожидания перехода в сигнальное состояние события SvcctrlStartEvent_A3752DX.

Затем функция SvcCtrlMain приступает к своей работе и вызывает ScGenerateServiceDB, функцию, создающую внутреннюю базу данных служб SCM. Функция ScGenerateServiceDB считывает и сохраняет содержимое, хранящееся в разделе HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder\List, в параметре REG_MULTI_SZ, где перечислены имена и порядок определения групп служб. Раздел реестра службы содержит необязательный параметр Group, если эта служба или драйвер нуждаются в управлении порядком своего запуска по отношению к службам из других групп. Например, сетевой стек Windows создается снизу вверх, поэтому сетевые службы должны указывать в параметрах Group, что в пусковой последовательности их нужно поместить позже драйверов сетевых устройств. SCM внутри себя создает список групп, помогающий упорядочить группы, считываемые им из реестра. В список групп входят NDIS, TDI, Primary Disk, Keyboard Port и Keyboard Class (и не только они). Дополнения и приложения сторонних разработчиков могут даже определить свои собственные группы и добавить их к списку. К примеру, сервер транзакций — Microsoft Transaction Server — добавляет группу MS Transactions.

Затем функция ScGenerateServiceDB сканирует содержимое раздела HKLM\SYSTEM\CurrentControlSet\Services, создавая запись в базе данных служб для

каждого найденного им раздела. Запись базы данных включает все параметры, связанные со службой, а также поля, связанные с состоянием службы. SCM добавляет записи для драйверов устройств, а также для служб, поскольку SCM запускает службы и драйверы, помеченные для автозапуска, и определяет сбои запуска тех драйверов, которые помечены для запуска в ходе загрузки и запуска системы. Он также предоставляет приложениям средства для запроса состояния драйверов. Диспетчер ввода-вывода загружает драйверы, помеченные для запуска в ходе загрузки и запуска системы до выполнения любых процессов пользовательского режима, и поэтому любые драйверы, имеющие эти типы запуска, загружаются перед запуском SCM.

Функция ScGenerateServiceDB считывает параметр службы Group, чтобы определить ее принадлежность к группе и связать этот параметр с записью группы в созданном ранее списке групп. Функция также считывает и записывает в базу данных зависимости службы и группы, запрашивая ее параметры реестра DependOnGroup и DependOnService. На рис. 4.11 показано, как SCM создает запись службы и списки порядка следования групп. Обратите внимание на то, что список служб отсортирован в алфавитном порядке. Причина такой сортировки состоит в том, что SCM создает список из раздела реестра Services, а Windows сохраняет разделы реестра в алфавитном порядке.

В ходе запуска службы SCM вызывает LSASS (например, для входа в службу под учетной записью нелокальной системы), поэтому SCM ждет от LSASS выдачи сигнала события синхронизации LSA_RPC_SERVER_ACTIVE, что этот процесс авторизации локальных пользователей делает по завершении инициализации. Wininit также запускает процесс LSASS, поэтому инициализация LSASS проходит одновременно с инициализацией SCM, и порядок, в котором LSASS и SCM завершают инициализацию, может изменяться. Затем SvcCtrlMain вызывает функцию ScGetBootAndSystemDriverState, чтобы просканировать базу данных служб в поиске записей драйверов устройств, запускаемых при загрузке и при запуске системы.

Функция ScGetBootAndSystemDriverState определяет, был ли драйвер успешно запущен, путем поиска имени в каталоге пространства имен диспетчера \Driver. Когда драйвер устройства успешно загружается, диспетчер ввода-вывода вставляет объект драйвера в пространство имен в этот каталог, поэтому если его имя отсутствует, он не был загружен. На рис. 4.12 показано, что WinObj выводит содержимое каталога Driver. Функция SvcCtrlMain отмечает имена незапущенных драйверов, и этот список становится частью текущего профиля в списке ScFailedDrivers.

Перед стартом служб, настроенных на автозапуск, SCM выполняет ряд дополнительных действий. Он создает свой вызов удаленной процедуры (RPC) именованного канала, который называет \Pipe\Ntsvcs, а затем RPC запускает поток для прослушивания канала на предмет входящих сообщений от SCP-программ. Затем SCM выставляет сигнал своему событию завершения инициализации, SvcctrlStartEvent_A3752DX. Регистрация обработчика события завершения работы консольного приложения и регистрация с помощью подсистемы Windows процесса через функцию RegisterServiceProcess подготавливают SCM к завершению работы системы.

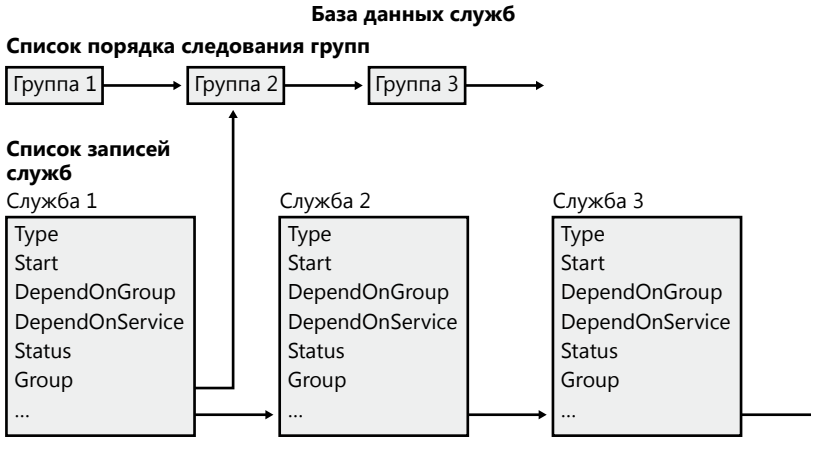


Рис. 4.11. Структура базы данных служб

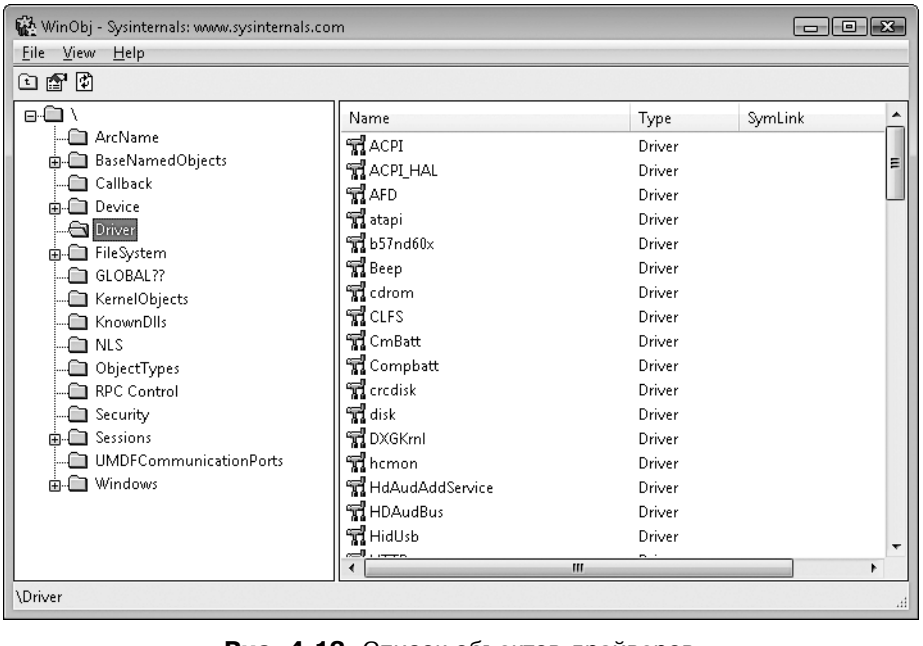


Рис. 4.12. Список объектов драйверов

БУКВЫ СЕТЕВЫХ ДИСКОВ

В дополнение к своей роли предоставления интерфейса к службам у SCM есть еще одна, совершенно не связанная с эти обязанность: он уведомляет GUI-приложения, работающие в системе, о создании или удалении системой подключений к букве сетевого диска. SCM ждет, когда маршрутизатор многосетевого доступа — Multiple Provider Router (MPR) — просигнализирует об именованном событии \BaseNamedObjects\ScNetDrvMsg. MPR сигнализирует, когда приложение назначает букву диска удаленной общей

сети или удаляет такое назначение (см. главу 7 раздел «Сети»). Когда MPR сигнализирует о событии, SCM вызывает Windows-функцию GetDriveType для запроса списка подключенных букв сетевых дисков. Если список изменяется по сигналу о событии, SCM отправляет общее Windows-сообщение типа WM_DEVICECHANGE. SCM использует в качестве подтипа сообщения либо DBT_DEVICEREMOVECOMPLETE, либо DBT_DEVICEARRIVAL. Это сообщение главным образом предназначено для Windows Explorer, чтобы он мог обновить любое открытое окно Компьютер (Computer), показав в нем присутствие или отсутствие той или иной буквы сетевого диска.

Запуск службы

Для старта всех служб, имеющих в параметре Start указание на автозапуск (кроме служб с отложенным автозапуском), функция SvcCtrlMain вызывает SCM-функцию ScAutoStartServices. Эта функция также запускает драйверы устройств, настроенные на автозапуск. Чтобы избежать путаницы, нужно уяснить, что понятие «службы» означает службы и драйверы, если не указано что-либо иное. Алгоритм в ScAutoStartServices, предназначенный для запуска служб в правильном порядке, реализуется поэтапно. Тот этап, который относится к группе, и последовательно реализуемые этапы определяются порядком следования групп, сохраненном в параметре реестра HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder\List. Параметр List, показанный на рис. 4.13, включает имена групп в том порядке, в котором SCM должен их запускать. Таким образом, назначение принадлежности службы к группе не имеет никакого эффекта помимо настройки ее запуска по отношению к другим службам, принадлежащим другим группам.

Рис. 4.13. Раздел реестра ServiceGroupOrder

Когда этап стартует, ScAutoStartServices помечает, что все записи служб принадлежат запускаемой группе этапа. Затем ScAutoStartServices осуществляет циклический перебор помеченных служб, чтобы посмотреть, можно ли запустить каждую из них. Часть этой проверки включает просмотр, не помечена ли служба как автозапускаемая, что заставляет SCM запустить ее на следующем этапе. (Службы с отложенным автозапуском также должны быть разгруппированы.) Еще одна часть проверки заключается в определении зависимости службы от другой группы, что указывается наличием параметра DependOnGroup в разделе реестра, относящемся к службе. Если зависимость существует, группа, от которой зависит служба, должна быть уже инициализирована, и как минимум одна служба из этой группы должна быть успешно запущена. Если служба зависит от группы,

которая в последовательности запуска групп запускается позже, чем группа этой службы, SCM отмечает для службы ошибку «циклической зависимости» (circular dependency). Функция ScAutoStartServices рассматривает службу Windows или драйвер устройства, настроенный на автозапуск, затем она проверяет зависимость службы от одной или нескольких других служб, и если таковая имеется, проверяет факт запуска этих служб. Зависимости служб показаны с помощью параметра DependOnService, который находится в разделе реестра, относящемся к службе. Если служба зависит от других служб, принадлежащих группе, которая следует в списке ServiceGroupOrder\List позже, SCM также генерирует ошибку «циклической зависимости» и не запускает службу. Если служба зависит от любых служб той же группы, которые еще не были запущены, она пропускается.

Когда зависимости службы будут удовлетворены, функция ScAutoStartServices проводит перед запуском службы заключительную проверку, не является ли служба частью текущей пусковой конфигурации. Когда система запускается в безопасном режиме, SCM убеждается в том, что служба помечена либо по имени, либо по группе в соответствующем разделе реестра, относящемся к безопасному запуску. В разделе HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot есть два подраздела безопасного запуска: минимальный — Minimal и сетевой — Network, и какой из них проверяет SCM, зависит от того режима безопасного запуска, который был выбран пользователем. Если пользователь в специальном меню загрузки выбрал безопасный режим — Safe Mode — или безопасный режим с поддержкой командной строки — Safe Mode With Command Prompt, — SCM обращается к разделу Minimal, а если пользователь выбрал безопасный режим с загрузкой сетевых драйверов — Safe Mode With Networking, — SCM обращается к разделу Network. Существование строкового параметра Option в разделе SafeBoot показывает не только то, что система была запущена в безопасном режиме, но также и тип безопасного режима, выбранный пользователем.

Как только SCM решит запустить службу, он вызывает функцию ScStartService, которая производит для служб и для драйверов устройств разные действия. Когда ScStartService запускает службу Windows, сначала определяется имя файла, запускающего процесс службы, путем считывания значения параметра ImagePath из раздела реестра, относящегося к службе. Затем проверяется значение параметра Type, и если это значение равно SERVICE_WINDOWS_SHARE_PROCESS (0x20), SCM убеждается в том, что процесс, в котором запускается служба, уже запущен и зарегистрирован с использованием той же учетной записи, которая указана для запускаемой службы. (Это делается для того, чтобы убедиться, что служба не настроена с неверной учетной записью, такой как LocalService, а настроена с путем к образу, указывающим на запущенный процесс Svchost, например netsvcs, который запускается под учетной записью LocalSystem.) В параметре реестра ObjectName, относящемся к службе, хранится та учетная запись пользователя, под которой должна быть запущена служба. Служба, не имеющая параметра ObjectName или ObjectName of LocalSystem, запускается под учетной записью локальной системы.

SCM проверяет, что процесс службы не был уже запущен под другой учетной записью. Для этого SCM смотрит, не имеется ли во внутренней базе данных SCM, которая называется базой данных образов (image database), записи с параметром службы ImagePath. Если в этой базе данных такой записи для параметра ImagePath нет, SCM ее создает. Когда SCM создает новую запись, он сохраняет имя учетной

записи, используемое для службы, и данные из параметра службы ImagePath. SCM требует от служб наличия параметра ImagePath. Если у службы нет параметра ImagePath, SCM сообщает об ошибке, утверждая, что он не может найти путь к службе и не может запустить эту службу. Если SCM обнаруживает существующую запись в базе данных образов с соответствующими данными ImagePath, он убеждается в том, что информация об учетной записи пользователя для запускаемой им службы та же самая, что и информация, сохраненная в записи базы данных. Процесс должен быть зарегистрирован только под одной учетной записью, и поэтому если для службы указано другое имя учетной записи, чем для другой службы, уже запущенной в том же самом процессе, SCM сообщает об ошибке.

Для регистрации службы, если это указано в ее конфигурации, и для запуска процесса службы SCM вызывает функцию ScLogonAndStartImage. SCM регистрирует службы, запущенные не под учетной записью System путем вызова LSASS-функции LogonUserEx. Обычно эта функция требует пароль, но SCM указывает LSASS, что пароль сохранен как «секрет» LSASS, относящийся к службе в разделе реестра HKLM\SECURITY\Policy\Secrets[1]. Когда SCM вызывает функцию LogonUserEx, он в качестве регистрации службы указывает тип регистрации, поэтому LSASS ищет пароль в подразделе Secrets с именем вида _SC_<*имя службы*>.

Когда SCP настраивает информацию о регистрации службы, SCM предписывает LSASS сохранять пароль регистрации в виде секрета, используя функцию LsaStorePrivateData. Если регистрация проходит успешно, функция LogonUserEx возвращает вызывавшему коду дескриптор маркера доступа. В Windows маркеры доступа используются для представления контекста безопасности пользователя, а затем SCM связывает маркер доступа с процессом, в котором реализуется служба.

После успешной регистрации SCM загружает информацию о профиле учетной записи, если она еще не загружена, для чего вызывает функцию LoadUserProfile DLL-библиотеки UserEnv (%SystemRoot%\System32\Userenv.dll). В параметре HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList\<*раздел профиля пользователя*>\ProfileImagePath содержится место на диске куста реестра, который функция LoadUserProfile загружает в реестр, создавая для службы информацию в разделе куста HKEY_CURRENT_USER.

Интерактивная служба должна открыть станцию WinSta0, но перед тем как функция ScLogonAndStartImage разрешит интерактивной службе получить доступ к WinSta0, она смотрит, установлено ли значение в параметре HKLM\SYSTEM\CurrentControlSet\Control\Windows\NoInteractiveServices. Администраторы устанавливают этот параметр для того, чтобы службы, помеченные как интерактивные, не могли отображать окна в консоли. Эту возможность желательно иметь в среде окружения автоматических серверов, где нет пользователей для того, чтобы ответить на появление в пользовательском интерфейсе сеанса 0 уведомления от интерактивных служб.

Следующим действием функция ScLogonAndStartImage приступает к запуску процесса, если он еще не был запущен (например, для другой службы). SCM запускает процесс в приостановленном состоянии, для чего использует Windows-

[1] Содержимое раздела SECURITY обычно не отображается, поскольку исходные настройки безопасности этого раздела разрешают доступ только из учетной записи System.

функцию CreateProcessAsUser. Затем SCM создает именованный канал, по которому он обменивается данными с процессом службы, и присваивает каналу имя \Pipe\Net\NtControlPipe*X*, где *X* — это номер, который увеличивается всякий раз, когда SCM создает канал. SCM возобновляет выполнение процесса службы с помощью функции ResumeThread и ждет от службы подключения к ее SCM-каналу. Если этот канал существует, в параметре реестра HKLM\SYSTEM\CurrentControlSet\Control\ServicesPipeTimeout определяется время, которое отводится SCM на ожидание вызова службой функции StartServiceCtrlDispatcher и подключения, пока SCM от этого не откажется, завершит процесс и придет к заключению, что запуск службы не удался. Если параметр ServicesPipeTimeout не существует, SCM использует значение времени ожидания по умолчанию, которое равно 30 секундам. Этот параметр времени ожидания SCM использует для связи со всеми своими службами.

Когда служба подключается по каналу к SCM, этот диспетчер отправляет службе команду на запуск. Если служба за период времени ожидания так и не даст положительный ответ на команду запуска, SCM отказывается от ее запуска и переходит к запуску следующей службы. Когда служба не отвечает на запрос на запуск, SCM не завершает процесс, как это делается в случае, если служба не вызывает функцию StartServiceCtrlDispatcher за отведенное ей на это время, а уведомляет об ошибке системный журнал событий (Event Log), показывая, что служба не смогла запуститься по истечении времени ожидания.

Если служба, запускаемая SCM с помощью вызова функции ScStartService, имеет в параметре реестра Type значение SERVICE_KERNEL_DRIVER или SERVICE_FILE_SYSTEM_DRIVER, то служба на самом деле является драйвером устройства, поэтому функция ScStartService вызывает для загрузки драйвера функцию ScLoadDeviceDriver. Эта функция разрешает загрузку привилегий безопасности драйвера для SCM-процесса, а затем вызывает службу ядра NtLoadDriver, передавая ей данные параметра ImagePath, который находится в разделе реестра, принадлежащем драйверу. В отличие от служб драйверы не нуждаются в указании параметра ImagePath, и если этот параметр отсутствует, SCM создает путевое имя образа, добавляя имя драйвера к строке %SystemRoot%\System32\Drivers\.

Функция ScAutoStartServices продолжает последовательный перебор служб, принадлежащих группе, пока либо не будут запущены все службы, либо не будут сгенерированы ошибки зависимостей. Таким образом, SCM применяет циклический перебор для автоматического выстраивания служб в группе в соответствии с зависимостями DependOnService. На первых циклических проходах SCM должен запустить службы, от которых зависят другие службы, пропуская зависимые службы до последующих проходов. Следует заметить, что SCM игнорирует параметры Tag для служб Windows, которые могут встретиться в подразделах раздела HKLM\SYSTEM\CurrentControlSet\Services. Параметры Tag используются диспетчером ввода-вывода, чтобы распределить по порядку запуск драйверов устройств в группе на драйверы, запускаемые во время загрузки системы, и на драйверы, запускаемые при запуске системы. Как только SCM завершит выполнение этапов для всех групп, перечисленных в параметре ServiceGroupOrder\List, он выполняет этап для служб, принадлежащих группам, не перечисленным в параметре, а затем выполняет заключительный этап для служб, не состоящих в группах.

После обработки служб с автозапуском SCM вызывает функцию ScInitDelayStart. Эта функция выстраивает в очередь отложенные рабочие элементы, связанные с рабочим потоком, отвечающим за обработку всех служб, пропущенных функцией ScAutoStartServices из-за того, что они были помечены как имеющие отложенный автозапуск. Этот рабочий поток будет выполнен после задержки. По умолчанию задержка составляет 120 секунд, но она может быть изменена путем создания параметра AutoStartDelay в разделе HKLM\SYSTEM\CurrentControlSet\Control. SCM выполняет такие же действия, как и при запуске служб с неотложенным автозапуском.

ПРИМЕЧАНИЕ

Если у службы с неотложенным автозапуском в качестве одной из ее зависимостей имеется служба с отложенным автозапуском, флаг отложенного автозапуска будет проигнорирован и служба будет запущена без промедлений, чтобы удовлетворить зависимость.

СЛУЖБЫ С ОТЛОЖЕННЫМ АВТОЗАПУСКОМ

Службы с отложенным автозапуском позволяют Windows справиться с возрастающим количеством служб, которые запускаются, когда пользователь входит в систему. Запуск этих служб затормаживает процесс загрузки системы и увеличивает для пользователя время появления реакции на его действия со стороны элементов рабочего стола. Конструкция служб с автозапуском была изначально предназначена для тех служб, которые требуются на раннем этапе процесса загрузки, поскольку от них зависит работа других служб. Хорошим примером может послужить служба RPC, от которой зависят все остальные службы. Другой пример использования связан с разрешением автоматического запуска таких служб, как Windows Update. Поскольку многие из служб с автозапуском относятся ко второй категории, превращение их в службы с отложенным автозапуском позволяет важным службам запускаться быстрее, а пользовательскому рабочему столу быстрее быть в готовности к работе, когда пользователь вошел в систему сразу же после ее загрузки. Кроме того, эти службы работают в фоновом режиме, который понижает приоритет их потока, ввода-вывода и памяти. Настройка службы на отложенный автозапуск требует вызова API-функции ChangeServiceConfig2. Вы можете проверить состояние флага службы, воспользовавшись ключом qc bits для sc.exe.

Когда заканчивается запуск всех служб с автозапуском и драйверов, а также настройка рабочего элемента отложенного автозапуска, SCM выставляет сигнал о событии \BaseNamedObjects\SC_AutoStartComplete. Это событие используется программой Windows Setup для оценки продвижения запуска в ходе установки.

Ошибки, возникающие при запуске

Если драйвер или служба в ответ на SCM-команду запуска сообщают об ошибке, реакция SCM определяется значением параметра ErrorControl, находящегося в разделе реестра, относящемся к службе. Если параметр ErrorControl имеет значение SERVICE_ERROR_IGNORE (0) или этот параметр не указан, SCM просто игнорирует ошибку и продолжает обработку запуска служб. Если значение

параметра ErrorControl равно SERVICE_ERROR_NORMAL (1), SCM записывает событие в системный журнал событий, где сообщается следующее: «Служба <*имя службы*> не смогла запуститься из-за следующей ошибки:». SCM включает в запись журнала событий текстовое представления кода ошибки Windows, которое служба возвращает SCM в качестве причины сбоя запуска.

Если об ошибке сообщает служба, у которой значение параметра ErrorControl равно SERVICE_ERROR_SEVERE (2) или SERVICE_ERROR_CRITICAL (3), SCM делает запись в журнале событий, а затем вызывает внутреннюю функцию ScRevertToLastKnownGood. Эта функция переключает конфигурацию системного реестра на версию, известную как последняя удачная, с которой система в последний раз была запущена успешно. Затем он перезапускает систему, используя системную службу NtShutdownSystem, которая реализована в исполняющей системе. Если система уже загружена с последней удачной конфигурацией, она просто перезапускается.

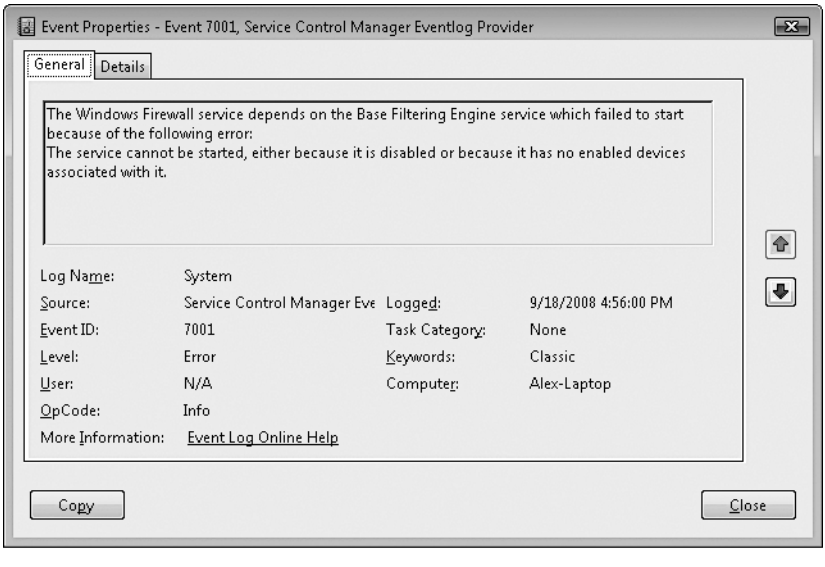


Рис. 4.14. Запись в журнале событий, касающаяся сбоя при запуске службы

Признание загрузки и последняя удачная конфигурация

Кроме запуска служб система нагружает SCM определением, когда конфигурация системного реестра, HKLM\SYSTEM\CurrentControlSet, должна быть сохранена в качестве последней удачной конфигурации. В разделе CurrentControlSet содержится подраздел Services, поэтому CurrentControlSet включает представление реестра в базе данных SCM. В нем также содержится подраздел Control, в котором содержатся настройки конфигураций многих подсистем режима ядра и пользовательского режима. По умолчанию последняя удачная конфигурация содержит успешные запуски служб с автозапуском и успешный вход пользователя в систему. Загрузка дает сбой при остановке системы, поскольку драйвер устройства приводит систему к аварии в ходе загрузки или в том случае, когда служба с ав-

тозапуском, имеющая значение параметра ErrorControl равное SERVICE_ERROR_SEVERE или SERVICE_ERROR_CRITICAL, сообщает об ошибке запуска.

SCM, конечно же, знает об успешном завершении запуска служб с автозапуском, но о том, насколько успешным был вход пользователя в систему, его должна уведомить программа Winlogon (%SystemRoot%\System32\Winlogon.exe). Эта программа при входе пользователя в систему вызывает функцию NotifyBootConfigStatus, и эта функция отправляет сообщение в адрес SCM. После успешного запуска служб с автозапуском или получения сообщения от функции NotifyBootConfigStatus (в зависимости от того, что было последним), SCM вызывает системную функцию NtInitializeRegistry, чтобы сохранить текущую пусковую конфигурацию реестра.

Разработчики стороннего программного обеспечения могут заменять даваемое программой Winlogon определение успешного входа в систему своим собственным определением. Например, система, запускающая Microsoft SQL Server, может не считать загрузку успешной, пока SQL Server не получит возможность принимать и обрабатывать транзакции. Разработчики могут назначать свое определение успешной загрузки путем написания программы проверки загрузки и установки, указывая ее местоположение на диске с параметром, хранящемся в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\BootVerificationProgram. Кроме того, установка программы проверки загрузки должна отключить вызов из Winlogon функции NotifyBootConfigStatus, установив значение параметра HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\ReportBootOk в 0. Когда программа проверки загрузки установлена, SCM запускает ее после завершения загрузки служб с автозагрузкой и перед сохранением последней удачной конфигурации ждет, когда программа вызовет функцию NotifyBootConfigStatus.

Windows сохраняет несколько копий текущего набора управления — CurrentControlSet, — и CurrentControlSet является просто символической ссылкой реестра, указывающей на одну из копий. У наборов управлений есть имена в виде HKLM\SYSTEM\ControlSet*nnn*, где *nnn* — это номера, например 001 или 002. В разделе HKLM\SYSTEM\Select содержатся параметры, идентифицирующие роль каждого набора управления. Например, если CurrentControlSet указывает на ControlSet001, параметр Current подраздела Select имеет значение 1. Параметр LastKnownGood подраздела Select содержит номер последнего удачного набора управления, то есть набора, который в последний раз использовался для удачной загрузки. Еще один параметр, который может находиться в вашей системе в разделе Select, называется Failed, и он указывает на последний набор управления, для которого загрузка была признана неудачной и была прервана с заменой на попытку загрузки с последним удачным набором управления. На рис. 4.15 показаны наборы управления системы и параметры раздела Select.

Функция NtInitializeRegistry берет содержимое последнего удачного набора управления и синхронизирует его с тем набором, который находится в дереве раздела CurrentControlSet. Если эта была первая удачная загрузка системы, последнего удачного набора управления не будет, и система создаст для него новый набор управления. Если дерево последнего удачного набора управления существует, система его обновляет, внося изменения между ним и CurrentControlSet.

Последний удачный набор управления приносит пользу в ситуациях, когда изменения, внесенные в CurrentControlSet (например, модификация параметра

настройки производительности системы в разделе HKLM\SYSTEM\Control или добавление службы или драйвера устройства), приводят к сбою при последующем запуске системы. Пользователь может нажать клавишу F8 на ранней стадии процесса загрузки для получения меню, позволяющего направить загрузку на использование последнего удачного набора управления, произвести откат конфигурации системного реестра к тому состоянию, в котором он был при последней успешной загрузке системы.

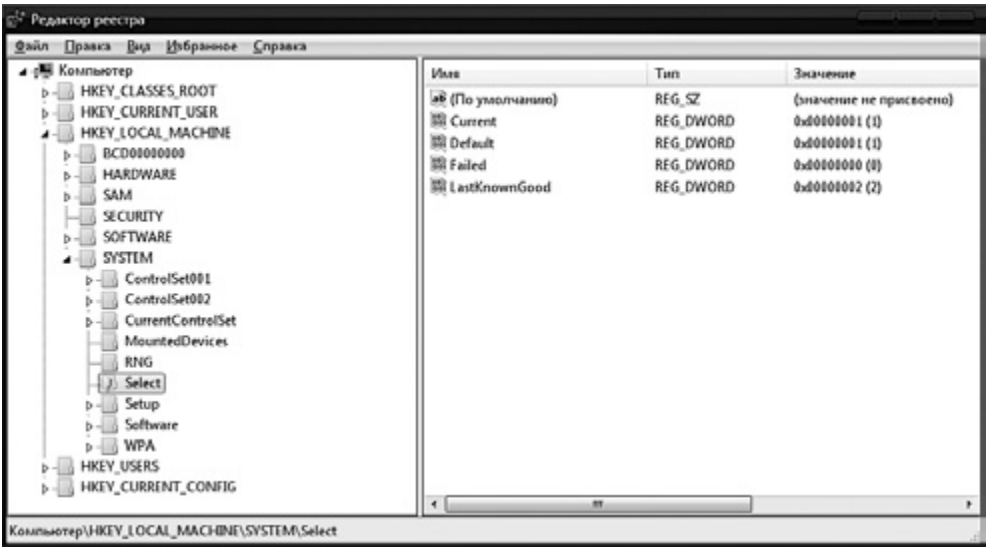


Рис. 4.15. Раздел выбора набора управления

Сбои служб

В разделах реестра, относящихся к службам, могут быть дополнительные параметры FailureActions и FailureCommand, куда SCM ведет запись в ходе запуска службы. SCM регистрируется с помощью системы, чтобы система сигнализировала SCM при выходе из процесса службы. При неожиданном завершении процесса службы SCM определяет, какие службы работали в процессе, и предпринимает восстановительные действия, указанные в параметрах реестра, относящихся к той или иной службе, и связанные с действиями в случае сбоев. Кроме того, службы не ограничены в запросах действий при сбоях только лишь в случае аварии или неожиданном завершении работы службы, поскольку к сбоям службы могут привести и другие проблемы, например дефицит оперативной памяти.

Если служба входит в состояние SERVICE_STOPPED, код ошибки, возвращенный SCM, не имеет значения ERROR_SUCCESS. SCM проверит, установлен ли у службы флаг действий, как при сбое, в случае отсутствия сбоев — FailureActionsOnNonCrashFailures, и выполнит те же восстановительные действия, как и при сбое службы. Чтобы воспользоваться этой функциональной возможностью, служба должна быть настроена с помощью API-функции ChangeServiceConfig2 или же системный администратор может воспользоваться утилитой Sc.exe с аргументом Failureflag для установки значения FailureActionsOnNonCrashFailures в 1. При зна-

чении по умолчанию, равном 0, SCM для всех остальных служб будет продолжать придерживаться того же самого поведения, что и в ранних версиях Windows.

Действия, которые служба должна настроить для SCM, включают перезапуск службы, запуск программы и перезагрузку компьютера. Кроме того, служба должна указать действия в случае сбоев (failure actions), которые выполняются при первом сбое процесса службы, при сбое во второй раз и при всех последующих случаях сбоев. Она может указать период задержки, в течение которого SCM находится в ожидании, перед тем как перезапустить службу, если служба того требует. Действия в случае сбоя службы IIS Admin Service заставляют SCM запускать приложение IISReset, которое наводит порядок и затем перезапускает службу. Как показано на рис. 4.16, восстановительные действия для службы легко поддаются настройке в MCC-оснастке Службы (Services), во вкладке Восстановление (Recovery) диалогового окна Свойства (Properties).

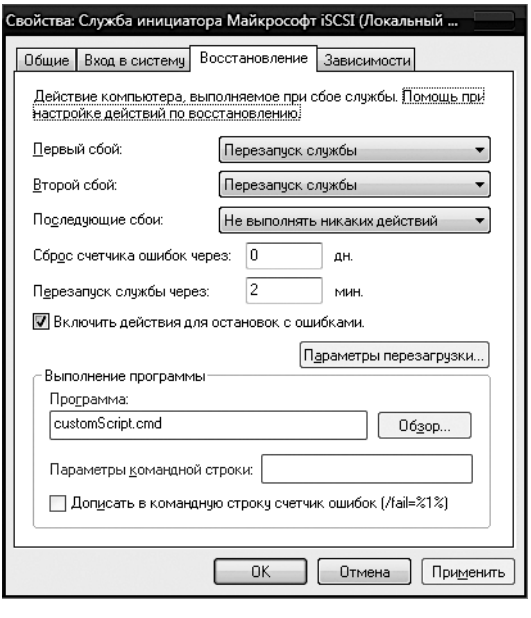


Рис. 4.16. Варианты восстановления работы службы

Остановка службы

Когда Winlogon вызывает Windows-функцию ExitWindowsEx, эта функция отправляет сообщение Csrss — процессу подсистемы Windows — для вызова Csrss-процедуры завершения работы. Csrss осуществляет циклический перебор активных процессов и уведомляет их, что система завершает работу. Для каждого системного процесса, за исключением SCM, Csrss ждет несколько секунд, количество которых определено параметром HKU\.DEFAULT\Control Panel\Desktop\WaitToKillAppTimeout (по умолчанию 20 с), пока не будет осуществлен выход из процесса, после чего переходит к следующему процессу. Когда Csrss встречает процесс SCM, он также уведомляет его о завершении работы системы, но использует лимит времени, определенный для SCM. Csrss распознает SCM, используя идентификатор процесса, который Csrss сохранил при регистрации

SCM с Csrss с помощью функции RegisterServicesProcess в ходе инициализации системы. Лимит времени для SCM отличается от лимита других процессов, поскольку Csrss знает, что SCM обменивается данными со службами, которым при остановке нужно осуществить подчистку после своей работы. Поэтому администратору может понадобиться настроить только лимит времени, выделяемый SCM. Параметр лимита времени SCM находится в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\WaitToKillServiceTimeout и по умолчанию составляет 12 секунд.

Обработчик остановки SCM отвечает за рассылку уведомлений всем службам, запросившим уведомление об остановке при своей инициализации с помощью SCM. SCM-функция ScShutdownAllServices осуществляет циклический перебор записей служб в базе данных SCM в поиске служб, желающих получить уведомление об остановке, и отправляет каждой из них команду на остановку. Для каждой службы, которой отправляется команда на остановку, SCM записывает параметр, указывающий время ожидания службы, значение которого служба также определяет при своей регистрации с помощью SCM. SCM отслеживает самое продолжительное указанное время ожидания. После отправки сообщений об остановке SCM ждет, либо пока одна из служб, которая была оповещена об остановке, не осуществит выход, либо пока не пройдет самое продолжительное указанное время.

Если указанное время истечет, а служба не осуществит выход, SCM определяет, не отправила ли одна или несколько служб, от которых ожидался выход, сообщение SCM о том, что эта служба находится в процессе своей остановки. Если хотя бы одна служба продвинулась в этом процессе, SCM снова ждет истечения установленного времени ожидания. SCM продолжает выполнение этого цикла ожидания, пока либо все службы не осуществят выход, либо ни одна из служб, в отношении которых осуществляется ожидание, не уведомит его о нахождении в процессе свой остановки за тот период времени, который указан для ожидания.

Пока SCM занимается отправкой приказов службам на остановку и ожиданием их выхода, Csrss ждет выхода от SCM. Если ожидание, осуществляемое Csrss, заканчивается до того, как будет осуществлен выход SCM (по истечении времени, указанного в параметре WaitToKillServiceTimeout), Csrss принудительно завершает работу SCM и продолжает процесс завершения работы системы. Таким образом, службы, давшие сбой при остановке за отведенное для этого время, останавливаются принудительно. Такая логика позволяет системе завершать работу вопреки тем службам, которые никогда не остановятся в результате дефектов своей конструкции, но это также означает, что службы, требующие более 20 секунд, не завершат своих действий по остановке.

Кроме того, поскольку порядок остановки ничем не обусловлен, службы, которые могут зависеть от других служб, для своей первоочередной остановки не имеют способа сообщить об этом SCM, могут не иметь никаких шансов на подчистку после своей работы.

В связи с этими потребностями в Windows реализуются предостановочные уведомления и порядок остановки для противодействия проблемам, вызываемым этими двумя сценариями. Предостановочные уведомления рассылаются с использованием того же механизма, который используется для уведомлений об остановке. Уведомления рассылаются тем службам, которые запросили предостановочные уведомления посредством API-функции SetServiceStatus, и SCM будет ждать от них подтверждения.

Идея, заложенная в этих уведомлениях, заключается в пометке тех служб, у которых на подчистку может уйти много времени (это, к примеру, может касаться служб сервера базы данных), и в выделении им большего количества времени на завершение их работы. SCM отправит запрос на выяснение хода процесса и будет 3 минуты ждать завершения работы тех служб, которые на него ответили. Если служба за это время не ответит, она будет принудительно остановлена в рамках процедуры завершения работы системы; в противном случае она может продолжать свое выполнение столько времени, сколько потребуется, пока она будет продолжать отвечать на запросы SCM.

Службы, участвующие в предварительной остановке (preshutdown), могут также определить порядок остановки по отношению к другим службам, также участвующим в предварительной остановке. Службы, зависящие от первоочередной остановки других служб (например, Group Policy должна дождаться завершения работы службы обновления Windows Update), могут указать свои зависимости от остановок других служб в параметре реестра HKLM\SYSTEM\CurrentControlSet\Control\PreshutdownOrder.

Процессы, общие для нескольких служб

Запуск каждой службы в ее собственном процессе вместо совместного использования процесса несколькими службами там, где это только возможно, распыляет системные ресурсы. Но совместное использование процесса означает, что наличие ошибки в какой-нибудь службе, запущенной в этом процессе, приведет к выходу из процесса с остановкой всех запущенных в нем служб.

Что касается служб, встроенных в Windows, то некоторые из них запускаются в своем собственном процессе, а некоторые делят процесс с другими службами. Например, процесс LSASS содержит службы, имеющие отношение к вопросам безопасности, такие как служба диспетчера учетных записей безопасности — Security Accounts Manager (SamSs), служба сетевого входа в систему — Net Logon (Netlogon) и служба изоляции ключей — Key Isolation (KeyIso) криптографии следующего поколения — Crypto Next Generation (CNG).

Есть также общий процесс Service Host (SvcHost–%SystemRoot%\System32\Svchost.exe), предназначенный для нескольких служб. В разных процессах может быть запущено несколько экземпляров SvcHost.

К службам, запускаемым в процессах SvcHost, относятся служба телефонии — Telephony (TapiSrv), удаленного вызова процедур — Remote Procedure Call (RpcSs) и диспетчера подключений удаленного доступа — Remote Access Connection Manager (RasMan). Windows реализует службы, запускаемые в SvcHost, в виде DLL-библиотек и включает определение параметра ImagePath в виде «%SystemRoot%\System32\svchost.exe –k netsvcs» в раздел реестра, относящийся к службе. В этом же разделе в подразделе Parameters должен также быть параметр ServiceDll, указывающий на DLL-файл службы.

Все службы, совместно использующие общий процесс SvcHost, указывают один и тот же параметр (в примере предыдущего абзаца это «–k netsvcs»), поэтому у них единая запись в базе данных образов SCM.

Когда SCM в ходе запуска служб сталкивается с первой службой, имеющей ImagePath SvcHost с особым параметром, он создает новую запись в базе данных

образов и запускает процесс SvcHost с параметром. Новый SvcHost-процесс получает параметр и ищет параметр с тем же именем, как и параметр в разделе HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost. SvcHost считывает содержимое параметра, интерпретирует его как список имен служб и уведомляет SCM, что он является хозяином этих служб, когда регистрируется с помощью SCM.

Когда SCM в ходе запуска служб сталкивается с SvcHost-службой (проверяя параметр типа службы) с параметром ImagePath, значение которого уже имеется в базе данных образов, он не запускает второй процесс, а вместо этого отправляет службе команду на запуск в SvcHost, который уже запущен для этого параметра ImagePath. Существующий процесс SvcHost считывает параметр ServiceDll в разделе реестра, относящегося к службе, и загружает DLL в свой процесс для запуска службы.

В табл. 4.11 перечислены все исходные группировки служб в Windows и некоторые, зарегистрированные для каждой из них.

Таблица 4.11. Главные группировки служб

Служба	Группа служб	Примечание
LocalService	Network Store Interface, Windows Diagnostic Host, Windows Time, COM+ Event System, HTTP Auto-Proxy Service, Software Protection Platform UI Notification, Thread Order Service, LLDT Discovery, SSL, FDP Host, WebClient	Службы, запускаемые под учетной записью локальной службы и пользующиеся сетью на различных портах или вообще не использующие сеть (и, следовательно, не имеющие ограничений)
LocalService-AndNoImpersonation	UPnP and SSDP, Smart Card, TPM, Font Cache, Function Discovery, AppID, qWAVE, Windows Connect Now, Media Center Extender, Adaptive Brightness	Службы, запускаемые под учетной записью локальной службы и пользующиеся сетью на фиксированном наборе портов. Службы работают с маркером доступа, ограничивающим запись
LocalService-Network-Restricted	DHCP, Event Logger, Windows Audio, NetBIOS, Security Center, Parental Controls, HomeGroup Provider	Службы, запускаемые под учетной записью локальной службы и пользующиеся сетью на фиксированном наборе портов
LocalService-NoNetwork	Diagnostic Policy Engine, Base Filtering Engine, Performance Logging and Alerts, Windows Firewall, WWAN AutoConfig	Службы, запускаемые под учетной записью локальной службы и вообще не пользующиеся сетью. Службы работают с маркером доступа, ограничивающим запись
LocalSystem-Network-Restricted	DWM, WDI System Host, Network Connections, Distributed Link Tracking, Windows Audio Endpoint, Wired/WLAN AutoConfig, Pnp-X, HID Access, User-Mode Driver Framework Service, Superfetch, Portable Device Enumerator, HomeGroup Listener, Tablet Input, Program Compatibility, Offline Files	Службы, запускаемые под учетной записью локальной службы и пользующиеся сетью на фиксированном наборе портов

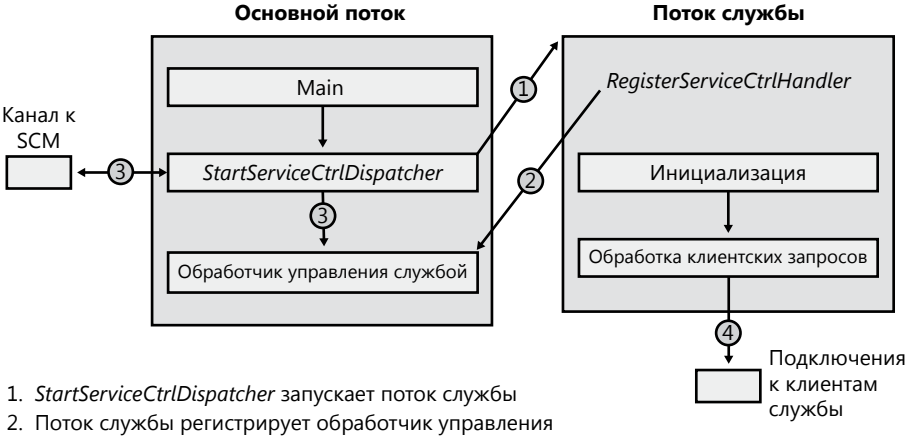
продолжение ⇩

Таблица 4.11 *(продолжение)*

Служба	Группа служб	Примечание
NetworkService	Cryptographic Services, DHCP Client, Terminal Services, WorkStation, Network Access Protection, NLA, DNS Client, Telephony, Windows Event Collector, WinRM	Службы, запускаемые под учетной записью сетевой службы и пользующиеся сетью на различных портах (или не имеющие накладываемых на них сетевых ограничений)
NetworkService-AndNoImperso-nation	KTM для DTC	Службы, запускаемые под учетной записью сетевой службы и пользующиеся сетью на фиксированном наборе портов. Службы работают с маркером доступа, ограничивающим запись
NetworkService-Network-Restricted	IPSec Policy Agent	Службы, запускаемые под учетной записью сетевой службы и пользующиеся сетью на фиксированном наборе портов

ЭКСПЕРИМЕНТ: ПРОСМОТР СЛУЖБ, ЗАПУЩЕННЫХ ВНУТРИ ПРОЦЕССА

Утилита Process Explorer дает подробную информацию о службах, запущенных внутри процессов. Запустите Process Explorer и просмотрите на вкладке Services (Службы) в диалоговом окне Process Properties (Свойства процесса) информацию для следующих процессов: Services.exe, Lsass.exe и Svchost.exe. На вашей системе будут запущены несколько экземпляров SvcHost и у вас появится возможность посмотреть на учетную запись, под которой запущен каждый экземпляр, если добавить к просмотру в Process Explorer столбец Username (Имя пользователя) или обратиться к полю Username (Имя пользователя) во вкладке Image (Образ), относящемуся к процессу диалогового окна Process Properties (Свойства процесса). На следующем экране показан список служб, запущенных в SvcHost, выполняющемся под учетной записью локальной службы.



1. *StartServiceCtrlDispatcher* запускает поток службы
2. Поток службы регистрирует обработчик управления
3. *StartServiceCtrlDispatcher* вызывает обработчики в ответ на команды SCM
4. Поток службы обрабатывает клиентские запросы

Выведенная информация включает в себя имя службы, отображаемое имя и описание, если таковое имеется, которое Process Explorer показывает под списком служб при выборе службы. Кроме того, показывается и путь к DLL-библиотеке, содержащей службу. Этой информацией можно воспользоваться для сопоставления стартовых адресов потоков (показанных на вкладке потоков — Threads) с их соответствующими службами, что может помочь в случаях возникновения проблем, связанных со службой, например, для выявления высокой загруженности центрального процессора.

Для просмотра списка служб, запущенных внутри процесса из окна командной строки, можно также воспользоваться инструментальным средством tlist.exe из комплекта Debugging Tools for Windows, или средством Tasklist, поставляемым с Windows. Для просмотра служб из Tlist можно использовать этот синтаксис:

```
tlist /s
```

А для просмотра из tasklist этот:

```
tasklist /svc
```

Учтите, что эти утилиты не показывают отображаемые имена служб или описания, они показывают только имена служб. ■

Теги служб

Один из недостатков использования процессов, в которых запускаются сразу несколько служб, заключается в существенном затруднении учета загруженности центрального процессора, а также учета затрат ресурсов для отдельной службы. Это связано с тем, что каждая служба совместно использует адресное пространство памяти, таблицу дескрипторов и учетные номера каждого процессора с другими службами, являющимися частью одной и той же группы служб. Хотя внутри хост-процесса служб всегда есть поток, принадлежащий конкретной службе, проследить связь потока со службой бывает нелегко. Например, служба для выполнения своих операций может использовать рабочий поток, или может быть так, что стартовый адрес и стек потока не раскрывают имя DLL-библиотеки, в которой находится служба, затрудняя разгадку того, какого рода работой может заниматься поток и какой службе он может принадлежать.

В Windows реализован атрибут службы, называемый ее тегом, который генерируется SCM путем вызова функции ScGenerateServiceTag при создании службы или когда в ходе загрузки системы генерируется база данных служб. Этот атрибут является просто индексом, идентифицирующим службу. Тег службы сохраняется в поле SubProcessTag блока окружения потока — thread environment block (TEB), — имеющегося у каждого потока (см. главу 5 «Процессы потоки и задания»), и распространяется на все потоки, создающиеся основным потоком службы (за исключением потоков, созданных опосредованно с помощью API-функций пула потоков).

Хотя теги служб хранятся внутри SCM, некоторые Windows-утилиты, например Netstat.exe, использует недокументированные API-функции для запроса тегов служб и отображения их на имена служб. Поскольку стек TCP/IP сохраняет тег службы того потока, который создал конечные точки TCP/IP, когда утилита Netstat запускается с ключом –b, она может показать имя службы для конечных точек, созданных службами. Еще одним инструментом, которым можно вос-

пользоваться для поиска тегов служб, является утилита ScTagQuery от Winsider Seminars & Solutions Inc. (www.winsiderss.com/tools/sctagquery/sctagquery.htm). Она может запросить SCM, чтобы тот отобразил тег каждой службы и показал эти теги либо для всей системы, либо для отдельных процессов. Она также может показать, каким службам принадлежат все потоки внутри хост-процесса служб. (Это обусловлено тем, что у этих потоков есть соответствующие теги связанных с ними служб.) Таким образом, если есть вышедшая из-под контроля служба, на которую тратится слишком много времени центрального процессора, можно идентифицировать виновную в этом службу, в случае если стартовый адрес потока или его стека не имеет явно связанной с ним DLL-библиотеки, содержащей службу.

Единый диспетчер фоновых процессов

За управление хозяйскими (hosted) или фоновыми (background) задачами, по мере роста сложности функций, традиционно отвечали разные компоненты Windows. В список этих компонентов входили от ранее рассмотренного диспетчера управления службами — Service Control Manager до планировщика задач — Task Scheduler, модуля запуска процессов DCOM-сервера — DCOM Server Launcher и WMI-поставщика — каждый из которых также отвечает за выполнение хозяйского (hosted) кода, выполняемого вне клиентского процесса (out-of-process). В настоящее время в Windows реализуется Единый диспетчер фоновых процессов — Unified Background Process Manager (UBPM), — который обрабатывает (по крайней мере, сейчас) два таких механизма (SCM и Task Scheduler), предоставляя этим компонентам возможность доступа к своим функциональным возможностям.

UBPM реализован в Services.exe, в том же месте, что и SCM, но в виде отдельной библиотеки, предоставляя свой собственный интерфейс через RPC (диспетчер устройств Plug and Play также запускается в Services.exe, но является отдельным компонентом). Он предоставляет доступ к интерфейсу через общедоступную экспортируемую DLL-библиотеку Ubpm.dll, которая открывается сторонним разработчикам служб через новые API-функции Trigger, которые были добавлены к SCM. Затем SCM загружает обычную библиотеку расширений SCM — SCM Extension DLL (Scext.dll), которая осуществляет вызовы в Ubpm.dll. Этот уровень опосредованности нужен для поддержки MinWin, где Scext.dll не загружается, и SCM предоставляет лишь минимальную функциональность. Эта архитектура изображена на рис. 4.17.

Инициализация

UBPM инициализируется с помощью SCM, когда его экспортируемая функция UbpmInitialize вызывается функцией ScExtInitializeTerminateUbpm в DLL-библиотеке SCM Extension. По существу, она реализована в виде DLL, запущенной в контексте SCM, а не в виде своего собственного отдельного процесса.

В начале своей инициализации UBPM настраивает библиотеку внутренних утилит. Благодаря многим усовершенствованиям в последних версиях Windows UBPM использует пул потоков для обработки множества поступающих событий, которые будут показаны позже, что позволяет ему масштабироваться от наличия одного рабочего потока до использования до 1000 таких потоков (что обуслов-

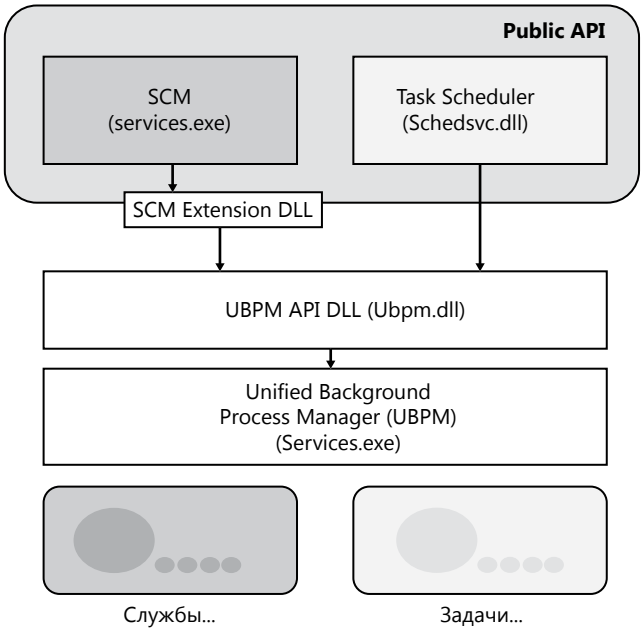


Рис. 4.17. Общий вид архитектуры UBPM

ливается максимально возможным количеством обрабатываемых потребителей, равным 10 000).

Затем UBPM инициализирует свою внутреннюю поддержку трассировки, которую можно настроить с помощью параметра Flags в разделе реестра HKLM\Soтware\Microsoft\Windows NT\CurrentVersion\Tracing\UBPM\Regular. Это может пригодиться для отладки и отслеживания поведения UBPM с использованием механизма трассировки WPP, описание которого дано в Windows Driver Kit.

Затем настраивается диспетчер событий, который будет использоваться более поздними компонентами UBPM для отчета о состоянии внутренних событий. Диспетчер событий регистрирует TASKSCHED GUID, который может использоваться событиями ETW, и записывает свое состояние в файл журнала TaskScheduler.log.

Следующим важным для UBPM шагом является инициализация его собственного ETW-потребителя реального времени, который является центральным механизмом, используемым UBPM для выполнения его работы, поскольку почти все получаемые им данные проходят как ETW-события. UBPM запускает ETW-сеанс реального времени в безопасном режиме (то есть будет единственным процессом, способным получать его события) и называет свой сеанс UBPM. Он также включает первого встроенного поставщика (владельцем которого является ядро) с целью получения уведомлений, связанных с изменениями времени.

Затем он связывает функцию обратного вызова события — UbpmpEventCallback — с поступающими событиями и создает потребительский поток, UbpmpConsumeEvents, который ожидает SCM-событие, используемое для оповещения о том, что событие автозапуска (названное ранее) завершено. Как только это происходит, потребительский поток вызывает функцию ProcessTrace,

которая совершает вызов в ETW и блокирует поток до тех пор, пока не завершится ETW-трассировка (обычно только при наличии UBPM). С другой стороны, имеющаяся в событии функция обратного вызова потребляет каждое ETW-событие по мере его поступления и обрабатывает его в соответствии с алгоритмом, который будет рассмотрен в следующем разделе.

ETW автоматически воспроизводит любые события, пропущенные до вызова ProcessTrace, это означает, что все события ядра в ходе загрузки системы сразу попадут в категорию поступающих и будут соответствующим образом обработаны. UBPM также ждет событие автозапуска SCM, которое гарантирует, что при поступлении события для него уже будет зарегистрировано по крайне мере две службы. В противном случае слишком раннее начало трассировки приведет к событиям без зарегистрированных потребителей и их утрате.

В завершение UBPM настраивает локальный RPC-интерфейс на TaskHost — второй компонент UBPM, который будет рассмотрен чуть позже, а тот, в свою очередь, также настраивает свой собственный локальный RPC-интерфейс, который открывает доступ к API-функциям, позволяющим службам использовать функциональные возможности UBPM (регистрация триггер-поставщиков, генерация триггеров и уведомлений и т. п.). Эти API-функции реализованы в библиотеке Ubpm.dll и используют RPC для связи RPC-интерфейса в коде UBPM файла Services.exe.

Когда происходит выход из UBPM, в обратном порядке выполняются все противоположные действия, чтобы восстановить предыдущее состояние системы.

API-функции UBPM

UBPM позволяет пользующимся своими API-функциями службам воспользоваться следующими механизмами:

- регистрирование и отмена регистрации триггер-поставщика, а также открытие и закрытие обработчика для него;
- генерация уведомления или триггера;
- настройка и запросы конфигурации триггер-поставщика;
- отправка команд управления триггер-поставщику.

Регистрация поставщика

Поставщики регистрируются через DLL-библиотеку SCM Extension, которая использует функцию ScExtpRegisterProvider, используемую функцией ScExtGenerateNotification. Тем самым открывается дескриптор UBPM и вызывается API-функция UbpmRegisterTriggerProvider. Когда служба регистрирует поставщика, она должна определить для него уникальное имя и GUID, а также флаги, необходимые для определения поставщика (например, используя флаг ETW-поставщика). Кроме этого у поставщика должно быть запоминающееся имя и описание. Когда регистрация завершится, поставщик будет внесен в список поставщиков UBPM, общее количество поставщиков увеличится на единицу, и если это ETW-поставщик, который не был запущен из-за установленного флага выключения, GUID поставщика будет включен в ETW-трассировку реального времени, активированную при инициализации. Созданный блок поставщика содержит всю информацию о поставщике, которая была взята при регистрации.

После того как поставщик будет зарегистрирован, для увеличения значения счетчика ссылок на поставщика и возвращения его блока поставщика могут использоваться API-функции открытия и закрытия. Кроме того, если поставщик не был зарегистрирован в выключенном состоянии и состоявшаяся на него ссылка была первой, его GUID будет включен в ETW-трассировку реального времени.

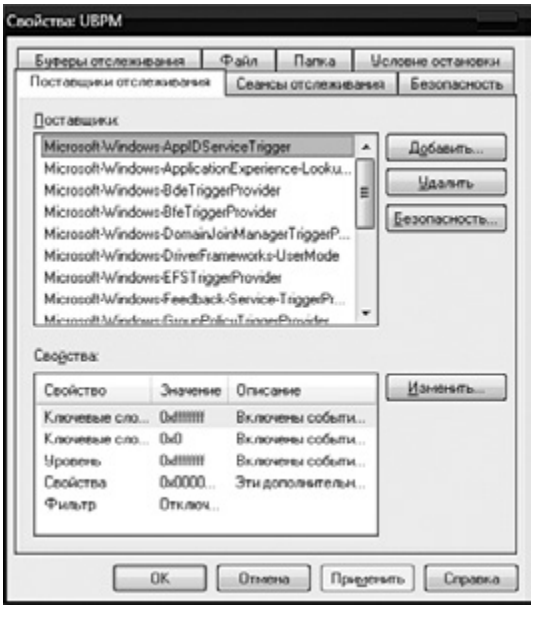
Аналогичным образом, не пройдя регистрацию, поставщик отключит свой GUID и уберет ссылку на него из списка поставщиков, и, как только ссылка будет закрыта, блок поставщика будет удален.

ЭКСПЕРИМЕНТ: ПРОСМОТР ТРИГГЕР-ПОСТАВЩИКОВ UBPM

Чтобы увидеть, что UBPM активно отслеживает всех ETW-поставщиков, которые зарегистрировались с его помощью, можно воспользоваться инструментальным средством Системный монитор (Performance Monitor). Для этого выполните следующие действия:

1. Откройте Системный монитор, щелкнув на кнопке Пуск (Start), после чего выберите пункт Выполнить (Run).
2. Наберите perfmon и щелкните на кнопке OK.
3. Когда запустится Системный монитор, щелчком на стрелке раскройте пункт Группы сборщиков данных (Data Collector Sets).
4. Выберите из списка пункт Сеансы отслеживания событий (Event Trace Sessions), а затем дважды щелкните на записи UBPM.

На следующем рисунке показаны поставщики триггеров UBPM на машине автора. У вас должна быть аналогичная картина.



Из длины списка следует, что зарегистрировались десятки поставщиков, каждый из которых способен генерировать индивидуальные события. Например, поставщик BfeTriggerProvider обрабатывает события брандмауэра. В последующем эксперимента вы увидите потребителя такого события. ■

Регистрация потребителя

Изначально регистрация потребителя службы предоставляется функцией обратного вызова ScExtRegisterTriggerConsumer, которая находится в DLL-библиотеке SCM Extension. Ее задача состоит в получении всей триггерной информации в формате SCM (в соответствии с документацией MSDN API «Service Trigger Events», доступной на сайте MSDN) и преобразования этой информации в простую структуру данных, используемую внутри UBPM. Когда вся обработка подойдет к концу, функция DLL-библиотеки SCM Extension DLL упаковывает триггер и связывает его с двумя действиями: запуска службы с помощью UBPM — UBPM Start Service и остановки службы с помощью UBPM — UBPM Stop Service.

Служба планировщика задач — Scheduled Tasks, — которая также использует UBPM, предоставляет аналогичные функциональные возможности посредством внутреннего синглтон-класса UBPM, который осуществляет вызов в Ubpm.dll. Это позволяет его внутренней API-функции RegisterTask также регистрироваться в качестве триггер-потребителя, при этом происходит аналогичная обработка его входных данных с той лишь разницей, что используется действие запуска исполняемого файла — UBPM Start EXE. Затем, чтобы выполнить фактическую регистрацию, открывается дескриптор UBPM и проверяется, не был ли потребитель уже зарегистрирован (внесение изменений в существующих потребителей не допускается), и в завершение поставщика регистрируется с помощью API-функции UbpmRegisterTriggerConsumer.

Регистрация потребителя триггера, произведенная функцией UbpmTriggerProviderRegister, которая проверяет запрос, приводит к добавлению GUID поставщика в список поставщиков и включает для него ETW-трассировку, чтобы теперь получать событие и об этом поставщике.

ЭКСПЕРИМЕНТ: ПРОСМОТР ТОГО, КАКИЕ СЛУЖБЫ РЕАГИРУЮТ НА КАКИЕ ТРИГГЕРЫ

Определенные Windows-службы уже заранее настроены на потребление соответствующих триггеров, чтобы предотвратить их нахождение в качестве резидентных даже когда они не нужны. Например, это относится к службам Windows Time Service, Tablet Input Service и Computer Browser Service. Команда sc позволяет запросить информацию о триггерах служб с помощью ключа qtriggerinfo.

1. Откройте окно командной строки.
2. Наберите следующую команду, чтобы увидеть триггеры для Windows Time Service:

```
sc qtriggerinfo w32time
[SC] QueryServiceConfig2 SUCCESS
SERVICE_NAME: w32time
        START SERVICE
          DOMAIN JOINED STATUS         : 1ce20aba-9851-4421-9430-1ddeb766e809
[DOMAIN JOINED]
        STOP SERVICE
          DOMAIN JOINED STATUS         : ddaf516e-58c2-4866-9574-c3b615d42ea1
[NOT DOMAIN JOINED]
```

3. Теперь посмотрите на Tablet Input Service:

```
sc qtriggerinfo tabletinputservice
[SC] QueryServiceConfig2 SUCCESS
SERVICE_NAME: tabletinputservice
        START SERVICE
          DEVICE INTERFACE ARRIVAL     : 4d1e55b2-f16f-11cf-88cb-001111000030
[INTERFACE CLASS GUID]
            DATA                       : HID_DEVICE_UP:000D_U:0001
            DATA                       : HID_DEVICE_UP:000D_U:0002
            DATA                       : HID_DEVICE_UP:000D_U:0003
            DATA                       : HID_DEVICE_UP:000D_U:0004
```

4. И наконец, на Computer Browser Service:

```
sc qtriggerinfo browser
[SC] QueryServiceConfig2 SUCCESS
SERVICE_NAME: browser
            START SERVICE
          FIREWALL PORT EVENT          : b7569e07-8421-4ee0-ad10-86915afdad09
    [PORT OPEN]
            DATA                       : 139;TCP;System;
            DATA                       : 137;UDP;System;
            DATA                       : 138;UDP;System;
        STOP SERVICE
          FIREWALL PORT EVENT          : a144ed38-8e12-4de4-9d96-e64740b1a524
[PORT CLOSE]
            DATA                       : 139;TCP;System;
            DATA                       : 137;UDP;System;
            DATA                       : 138;UDP;System;
```

В этих трех случаях обратите внимание на то, что Windows Time Service ждет присоединения к домену или выхода из него, чтобы решить нужно ли ей запускаться или нет, а Tablet Input Service ждет устройства с HID Class ID, который бы соответствовал Tablet Device. И наконец, Computer Browser Service запустится, только если политика брандмауэра позволит получить доступ к портам 137, 138 и 139, являющимся сетевыми портами SMB, нужными браузеру. ■

Task Host

Хост-процесс для задач Windows — TaskHost — получает команды от UBPM, который находится в SCM. В ходе инициализации он открывает локальный RPC-интерфейс, созданный UBPM во время его инициализации, и входит в бесконечный цикл, ожидая поступления команд через канал. На данный момент поддерживаются четыре команды, отправляемые через RPC API-функцию Task HostSendResponseReceiveCommand:

- остановка хоста;
- запуск задачи;

- остановка задачи;
- завершение задачи.

Кроме того, задачи хост-процесса снабжаются RPC API-функцией TaskHostReportTaskStatus, которая позволяет им уведомлять UBPM о состоянии их текущего выполнения при каждом вызове функции UbpmReportTaskStatus.

Все основанные на задачах команды фактически внутренне реализованы универсальной библиотекой COM Task, и, по существу, они приводят к созданию или уничтожению COM-компонентов.

Программы управления службами

Программы управления службами (SCP) являются стандартными приложениями Windows, использующими функции управления службами, имеющиеся у SCM, включая следующие функции:

- создания службы — CreateService;
- открытия службы — OpenService;
- запуска службы — StartService;
- управления службой — ControlService;
- запроса состояния службы — QueryServiceStatus;
- удаления службы — DeleteService.

Для использования SCM-функций SCP должна сначала открыть канал обмена данными с SCM, вызвав функцию OpenSCManager. Ко времени вызова функции открытия канала SCP должна указать тип действий, которые ей нужно выполнять. Например, если SCP просто нужно подсчитать и вывести количество служб, присутствующих в базе данных SCM, она в своем вызове функции OpenSCManager требует доступа к службе подсчета. В ходе своей инициализации SCM создает внутренний объект, представляющий базу данных SCM, и использует функции безопасности Windows для защиты объекта дескриптором безопасности, который указывает, под какой учетной записью можно открыть объект и с какими разрешениями на доступ к нему. Например, дескриптор безопасности показывает, что группа Authenticated Users может открыть SCM-объект с доступом к нему службы подсчета. Но открыть объект с доступом, требуемым для создания или удаления записи о службе, может только администратор.

Точно так же, как это делается для базы данных SCM, диспетчер SCM реализует меры безопасности для самих служб. Когда SCP создает службу, используя функцию CreateService, она указывает дескриптор, который SCM внутренне связывает с записью службы в базе данных служб. SCM хранит дескриптор безопасности в параметре Security раздела реестра, относящегося к службе, и он считывает этот параметр при сканировании раздела реестра Services в ходе инициализации, чтобы установки безопасности существовали после перезагрузки. Точно так же, как SCP должна в своем вызове функции OpenSCManager указать, какого типа доступ ей нужен к базе данных SCM, SCP должна сообщить SCM в вызове функции OpenService, какой доступ ей нужен к службе.

Виды доступов, которые могут быть запрошены со стороны SCP, включают в себя возможность запроса состояния службы и настройки, остановки и запуска службы.

Программа SCP, с которой вы, вероятно, знакомы лучше всего, — это MMC-оснастка Службы (Services), включенная в Windows, которая находится в библиотеке %SystemRoot%\System32\Filemgmt.dll. В Windows также включена программа Sc.exe (Service Controller tool), программа командной строки, управляющая службами, о которой уже неоднократно упоминалось в этой книге.

Иногда SCP-программы выводят уровень политики службы на вершину того, что реализует SCM. Хорошим примером этому может послужить время ожидания, которое MMC-оснастка Службы (Services) реализует при ручном запуске службы. Оснастка показывает индикатор выполнения запуска службы. Службы опосредованно взаимодействуют с SCP-программами, устанавливая состояние своей конфигурации для отображения хода их ответа на команды SCM, такие как команда на запуск. SCP-программы запрашивают состояние с помощью функции QueryServiceStatus. Они могут отличать активное изменение состояния службы от ее зависания, и SCM может предпринимать соответствующие действия по уведомлению пользователя о том, чем занята служба.

Windows Management Instrumentation

Windows Management Instrumentation (WMI) является реализацией системы управления предприятием на основе использовании веб-технологии Web-Based Enterprise Management (WBEM), стандарта, определенного промышленным консорциумом Distributed Management Task Force (DMTF). Стандарт WBEM включает в себя разработку расширяемых средств сбора данных и управления данными предприятия, обладающих гибкостью и возможностями наращивания, требуемыми для управления локальными и удаленными системами, включающими в себя произвольные компоненты.

Архитектура WMI

Как показано на рис. 4.18, WMI состоит из четырех основных компонентов: приложений управления, инфраструктуры WMI, поставщиков и управляемых объектов. Приложения управления являются Windows-приложениями, которые имеют доступ к данным управляемых объектов и отображают или обрабатывают эти данные. Простым примером приложения управления является подстановка инструментария определения производительности, который зависит при получении информации о производительности от WMI, а не от API-функций определения производительности. Более сложным примером может послужить инструментарий управления предприятием, позволяющий администраторам выполнять автоматизированный учет программной и аппаратной конфигурации каждого компьютера предприятия.

Как правило, разработчики должны нацеливать приложения управления на сбор данных от конкретных объектов и на управление этими объектами. Объект должен представлять собой один компонент, такой как устройство сетевого адаптера, или коллекцию компонентов, такую как компьютер. (Объект компьютера может содержать объект сетевого адаптера.) Поставщики должны определять и экспортировать представления объектов, которыми интересуются приложения управления. Например, производителю сетевого адаптера может понадобиться добавить характерные для адаптера свойства к включенной в Windows WMI-

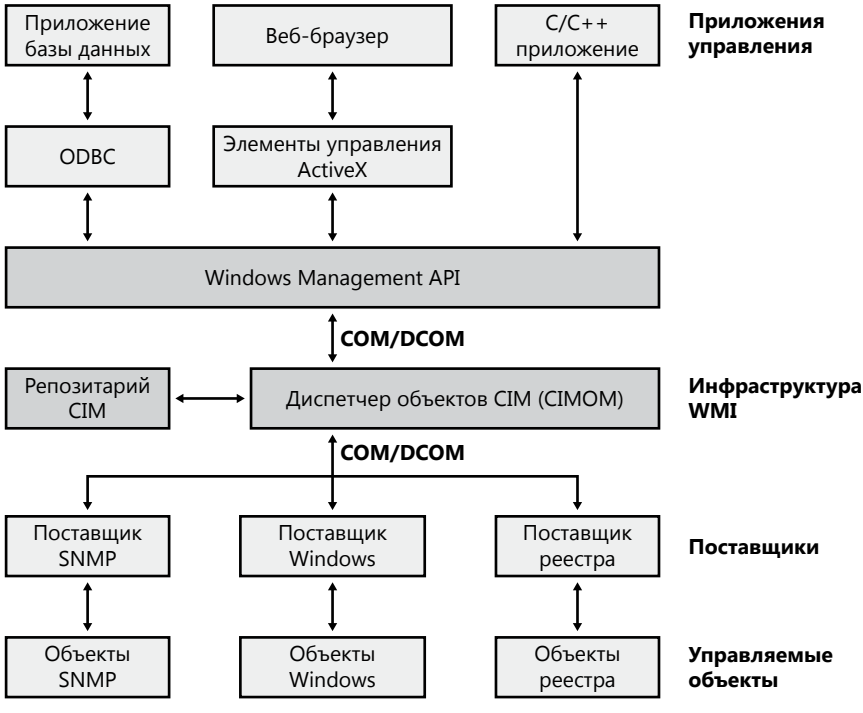


Рис. 4.18. Архитектура WMI

поддержке сетевого адаптера, запрашивая и устанавливая состояние и поведение адаптера по указанию приложения управления. В некоторых случаях (например, для драйверов устройств) Microsoft предоставляет поставщика, имеющего свои собственные API-функции, помогающие разработчикам использовать реализацию поставщика для своих собственных управляемых объектов с минимальными усилиями по программированию.

Инфраструктура WMI, основой которой служит CIMOM — диспетчер объектов (OM) общей информационной модели — Common Information Model (CIM), связующий компонент между приложениями управления и поставщиками. Инфраструктура служит также хранилищем объектов и классов и во многих случаях диспетчером хранилища для свойств постоянных объектов. WMI реализует хранилище, или репозиторий, в виде базы данных на диске, которая называется репозиторием объектов CIMOM. В качестве части своей инфраструктуры WMI поддерживают несколько API-функций, благодаря которым приложения управления получают доступ к данным объекта, а поставщики предоставляют данные и определения классов.

Для непосредственного взаимодействия с WMI программы и сценарии Windows (такие как Windows PowerShell) используют API-функции WMI COM, которые являются основными управляющими API-функциями. Другой уровень API-функций лежит на вершине COM API и включает в себя адаптер открытого интерфейса взаимодействия с базами данных — Open Database Connectivity (ODBC) для приложения базы данных Microsoft Access. Разработчиком базы данных адаптер

WMI ODBC используется для вставки ссылок на данные объекта в базу данных разработчика. Благодаря этому разработчик может легко генерировать отчеты с помощью запросов, содержащих данные на основе WMI. Управляющие элементы WMI ActiveX поддерживают API-функции другого уровня. Веб-разработчики используют элементы управления ActiveX для создания интерфейсов к WMI-данным на основе веб-технологий. Еще одним набором управляющих API-функций являются WMI API-функции для создания сценариев, которые используются в приложениях на основе сценариев и в программах, написанных на Microsoft Visual Basic. Поддержка WMI-функции для создания сценариев существует для всех технологий языков программирования Microsoft.

Поскольку они предназначаются для приложений управления, WMI COM-интерфейсы составляют основную часть API-функций для поставщиков. Но, в отличие от приложений управления, являющихся COM-клиентами, поставщики являются COM-серверами или распределенными COM-серверами (Distributed COM, DCOM). Возможные воплощения WMI-поставщика включают DLL-библиотеки, загружаемые в процесс WMI-диспетчера или автономные Windows-приложения или Windows-службы. Microsoft включает в себя несколько встроенных поставщиков, которые представляют данные от известных источников, таких как API-функции определения производительности (Performance API), реестр, диспетчер событий (Event Manager), Active Directory, SNMP и современные драйверы устройств. WMI SDK позволяет разработчикам создавать сторонних WMI-поставщиков.

Поставщики

В основу WBEM положена CIM-спецификация, разработанная DMTF. CIM определяет способ представления системы управления, то есть, с точки зрения управления системами, все, от компьютера до приложения или устройства на компьютере. Разработчик поставщика использует CIM для представления компонентов, составляющих те части приложения, которые разработчики хотят сделать управляемыми. Для реализации CIM-представления разработчики используют язык Managed Object Format (MOF).

Кроме определения классов, представляющих объекты, поставщик должен создавать интерфейс между WMI и объектами. WMI классифицирует поставщиков в соответствии со свойствами интерфейса, предоставляемого поставщиками. Классификация WMI-поставщиков перечислена в табл. 4.12. Следует учесть, что поставщик может реализовать одно или несколько свойств, поэтому он может быть, к примеру, как поставщиком класса, так и поставщиком события. Чтобы уточнить определения свойств в табл. 4.12, рассмотрим поставщика, реализующего несколько таких свойств. Поставщик журнала событий — Event Log — поддерживает несколько объектов, включая Event Log Computer, Event Log Record и Event Log File. Event Log является поставщиком экземпляра (Instance provider), потому что он может определять несколько экземпляров для нескольких своих классов. Одним из классов, для которого поставщик Event Log определяет несколько экземпляров, является класс Event Log File (Win32_NTEventlogFile). Поставщик Event Log определяет экземпляр этого класса для каждого журнала событий (то есть для журнала системных событий — System Event Log, журнала событий приложений — Application Event Log и журнала событий безопасности — Security Event Log).

Поставщик Event Log определяет данные экземпляра и позволяет управляющим приложениям провести подсчет записей. Чтобы позволить управляющим приложениям использовать WMI для создания резервной копии и восстановления файлов журнала событий, поставщик Event Log реализует для объектов Event Log File методы создания резервной копии (backup) и восстановления (restore). В результате этого поставщик Event Log является поставщиком методов (Method provider). И наконец, управляющее приложение может зарегистрироваться для получения уведомлений в случае появления новой записи в одном из журналов событий. Таким образом, когда поставщик Event Log использует WMI уведомление о событии, сообщающем WMI о поступлении записей в журнал событий, он служит в качестве поставщика событий (Event provider).

Таблица 4.12. Классификация поставщиков

Классификация	Описание
Class (поставщик классов)	Может предоставлять, изменять, удалять и подсчитывать определенные поставщиком классы. Может также поддерживать обработку запросов. Редкий пример службы, являющейся поставщиком классов, — Active Directory
Instance (поставщик экземпляров)	Может предоставлять, изменять, удалять и подсчитывать экземпляры классов, как системных, так и определенных поставщиком. Экземпляр представляет собой управляемый объект. Может также поддерживать обработку запросов
Property (поставщик свойств)	Может предоставлять и изменять значения свойств отдельного объекта
Method (поставщик методов)	Предоставляет методы для характерного для поставщика класса
Event (поставщик событий)	Генерирует уведомления о событиях
Event consumer (поставщик потребителей событий)	Отображает физического потребителя на логического потребителя для поддержки уведомлений о событиях

Common Information Model и язык Managed Object Format

Общая информационная модель — Common Information Model (CIM) — пошла по пути таких объектно-ориентированных языков, как C++ и C#, в которых разработчики модели придумали представления в виде классов. Работа с классами позволяет разработчикам использовать эффективную технологию моделирования, использующую наследование и составление. Подклассы могут наследовать свойства родительского класса и могут добавлять свои собственные характеристики и заменять характеристики, унаследованные у родительского класса. Класс, унаследовавший свойства у другого класса, происходит от этого класса. Классы можно также составлять: разработчик может создать класс, включающий другие классы.

DMTF предоставляет несколько классов, являющихся частью стандарта WBEM. Эти классы являются базовым языком CIM и представляют собой объекты, применимые ко всем областям управления. Классы — это часть основной

модели CIM. Примером основного класса является CIM_ManagedSystemElement. Этот класс содержит несколько базовых свойств, идентифицирующих такие физические компоненты, как аппаратные устройства, и логические компоненты, такие как процессы и файлы. Свойства включают заголовок, описание, дату установки и статус. Таким образом, классы CIM_LogicalElement и CIM_PhysicalElement наследуют свойства класса CIM_ManagedSystemElement. Эти два класса также являются частью основной модели CIM. В стандарте WBEM эти классы называются абстрактными классами, поскольку они существуют исключительно как классы, наследуемые другими классами (то есть экземпляров объектов абстрактных классов не существует). Поэтому абстрактные классы можно считать шаблонами, определяющими свойства для использования другими классами.

Вторая категория классов представляет объекты, характерные для областей управления, но не зависящие от конкретной реализации. Эти классы составляют общую модель и рассматриваются, как расширение основной модели. Примером класса общей модели может послужить класс CIM_FileSystem, наследующий свойства класса CIM_LogicalElement. Поскольку практически каждая операционная система, включающая Windows, Linux и другие разновидности UNIX, зависит от хранилища данных, основанного на файловой системе, класс CIM_FileSystem является вполне подходящей составляющей общей модели.

И заключительной категорией классов является расширенная модель, включающая в себя характерные для той или иной технологии дополнения к общей модели. В Windows определяется большой набор таких классов, чтобы представлять объекты, характерные для среды окружения Windows. Поскольку все операционные системы хранят данные в файлах, общая модель CIM включает класс CIM_LogicalFile. Класс CIM_DataFile наследует свойства класса CIM_LogicalFile, а Windows для соответствующих типов файлов добавляет классы файлов Win32_PageFile и Win32_ShortcutFile.

Наследование широко используется поставщиком Event Log. На рис. 4.19 показан вид WMI CIM Studio, браузера классов, поставляемого вместе с инструментарием WMI Administrative Tools, который можно получить в центре загрузки (Download Center) Microsoft на веб-сайте этой компании. На рисунке видно, что поставщик Event Log зависит от наследования из класса поставщика Win32_NTEventlogFile, который происходит из класса CIM_DataFile. Файлы Event Log являются файлами данных, которые имеют такие характерные для журнала событий дополнительные свойства, как имя файла журнала (LogfileName) и счетчик количества записей, содержащихся в файле (NumberOfRecords). Из дерева, показанного браузером классов, следует, что Win32_NTEventlogFile основан на нескольких уровнях наследования, где класс CIM_DataFile происходит от CIM_LogicalFile, который происходит от CIM_LogicalElement, а CIM_LogicalElement происходит от CIM_ManagedSystemElement.

Как уже упоминалось, разработчики WMI-поставщиков создают свои классы на языке MOF. Следующий вывод данных показывает определение относящегося к проводнику Event Log класса Win32_NTEventlogFile, выбранного на рис. 4.19. Обратите внимание на взаимосвязь между свойствами в списке свойств на правой панели рис. 4.19 и определениями этих свойств в следующем MOF-файле. Чтобы указать свойства, унаследованные классом, в CIM Studio используются желтые стрелки. Поэтому указаний на эти свойства в определении Win32_NTEventlogFile не видно.

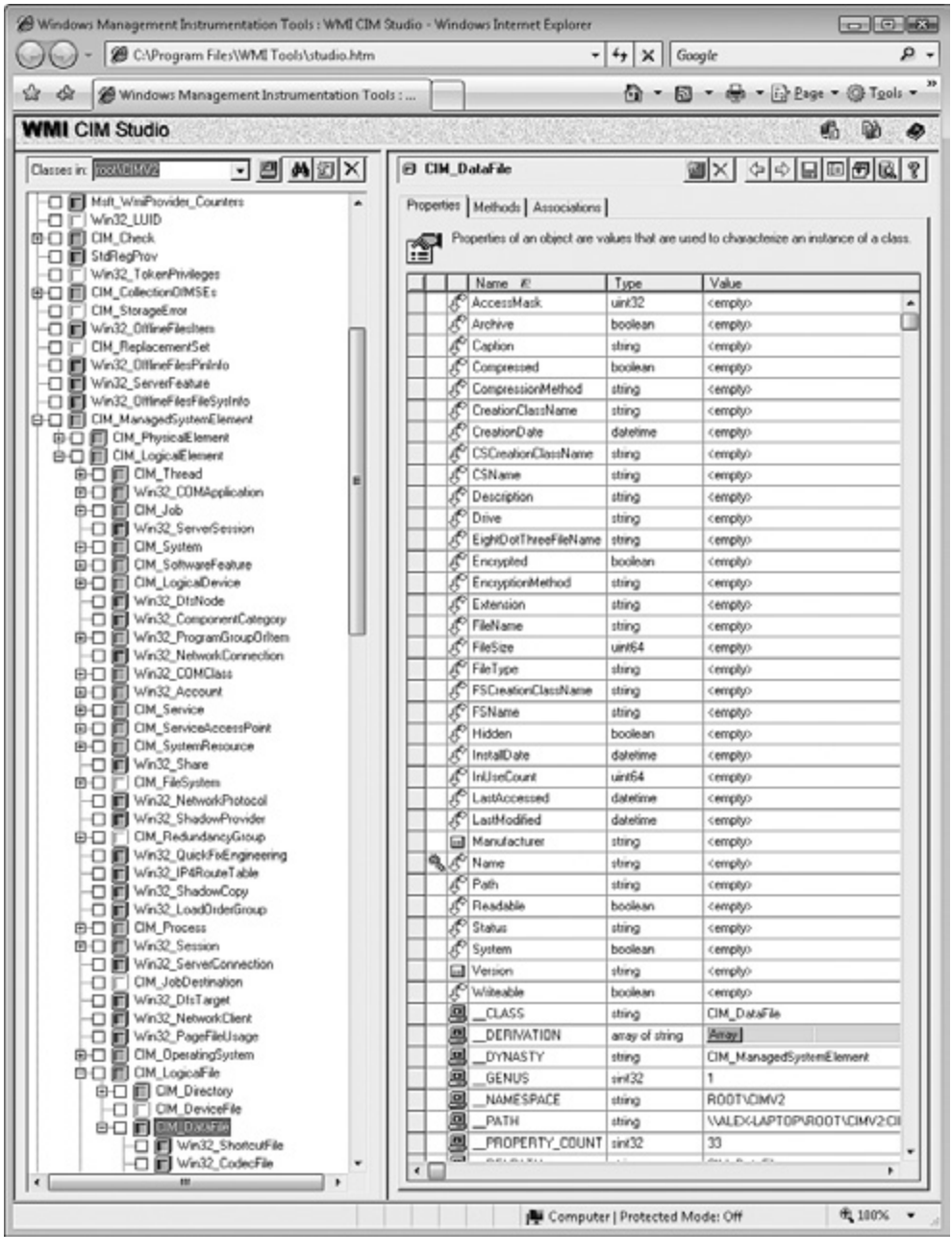


Рис. 4.19. WMI CIM Studio

Одним из понятий, заслуживающих рассмотрения, является динамика, наглядный показатель для класса Win32_NTEventlogFile, фигурирующего в предыдущем выводе данных MOF-файла. «Динамика» здесь означает, что WMI-инфраструктура запрашивает у WMI-поставщика значения свойств, связанных

с объектом класса, при каждом запросе свойств объекта со стороны приложения управления. Статический класс входит в классы, хранящиеся в репозитории WMI, а WMI-инфраструктура ссылается на репозиторий для получения значений вместо их запросов у поставщика. Поскольку обновление репозитория является относительно затратной операцией, динамичные поставщики более эффективны для объектов, имеющих часто изменяющиеся свойства.

```
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);
};
```

ЭКСПЕРИМЕНТ: ПРОСМОТР MOF-ОПРЕДЕЛЕНИЙ WMI-КЛАССОВ

MOF-определение любого WMI-класса можно просмотреть с помощью инструментального средства WbemTest, поставляемого вместе с Windows. В данном эксперименте будет найдено MOF-определение класса Win32_NTEventLogFile:

1. Запустите Wbemtest из диалогового окна Выполнить (Run), открываемого в меню Пуск (Start).
2. Щелкните на кнопке Подключить (Connect), измените Пространство имен (Namespace) на root\cimv2 и подключитесь.
3. Щелкните на кнопке Классы (Enum Classes), установите переключатель в положение Рекурсивное вхождение (Recursive option) и щелкните на кнопке OK.
4. Найдите в списке классов Win32_NTEventLogFile, а затем дважды щелкните на его названии, чтобы посмотреть на свойства этого класса.
5. Щелкните на кнопке Вывести MOF (Show MOF), чтобы открыть окно, в котором выводится текст MOF. ■

После создания классов в MOF WMI-разработчики могут предоставить WMI определения класса несколькими способами. Разработчики драйверов WDM компилируют MOF-файл в его двоичный вид, в файл binary MOF (BMF), более компактное по сравнению с MOF-файлом представление, и могут выбрать динамическое предоставление BMF-файлов в WDM-инфраструктуре или статическое включение его в ее двоичный файл. Еще один способ заключается в компиляции

поставщиком MOF-файла и использовании API-функций WMI COM для предоставления определения WMI-инфраструктуре. И наконец, поставщик может воспользоваться MOF-компилятором (Mofcomp.exe), чтобы дать WMI-инфраструктуре представление скомпилированных классов непосредственным образом.

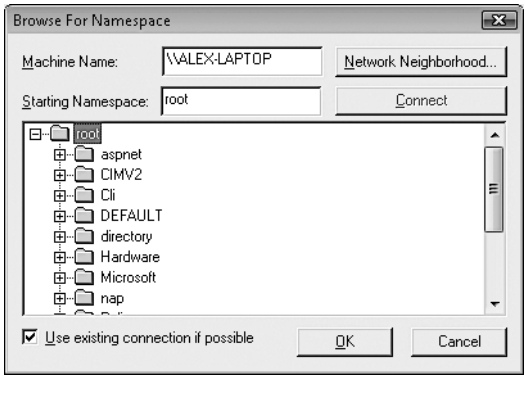
Пространство имен WMI

Классы определяют свойства объектов, а объекты являются экземплярами класса в системе. В WMI используется пространство имен, содержащее несколько подчиненных пространств имен, которые WMI для организации объектов выстраивает в иерархической структуре. Перед обращением к объектам в пространстве имен приложения управления должны подключиться к этому пространству.

Корневой каталог пространства имен WMI называет каталогом root. У всех WMI-установок имеются четыре предопределенные пространства имен, размещающиеся в каталоге root: CIMV2, Default, Security и WMI. Некоторые из этих пространств имеют внутри себя другие пространства имен. Например, CIMV2 включает в себя в качестве подчиненных пространства имен Applications и ms_409. Иногда поставщики определяют свои собственные пространства имен. Пространство имен WMI (которое определяет WMI-поставщик драйверов устройств Windows) можно увидеть в Windows ниже каталога root.

ЭКСПЕРИМЕНТ: ПРОСМОТР ПРОСТРАНСТВ ИМЕН WMI

Посмотреть, какие пространства имен определены в системе, можно с помощью WMI CIM Studio. Это средство предоставляет при запуске диалоговое окно подключения, включающее кнопку просмотра пространства имен справа от поля редактирования пространства имен. Открытие браузера и выбор пространства имен заставляет WMI CIM Studio подключиться к этому пространству имен. В Windows определяются свыше десятка пространств имен под каталогом root.



В отличие от пространства имен файловой системы, которое содержит иерархию каталогов и файлов, пространство имен WMI имеет глубину всего в один уровень. Вместо использования таких имен, как в файловой системе, в WMI используются свойства объектов, определенные в качестве ключей для идентификации объектов. Приложения управления определяют имена классов с ключевыми именами для определения местонахождения конкретных объектов в пространстве

имен. Таким образом, каждый экземпляр класса должен идентифицироваться уникальным образом по своим ключевым значениям. Например, поставщик Event Log для представления записей в журнале событий использует класс Win32_NTLogEvent. У этого класса есть два ключа: один, Logfile, в виде строки, а второй, RecordNumber, в виде целого числа без знака. Приложение управления, запрашивающее у WMI экземпляры записей журнала событий, получают их у поставщика ключевых пар, идентифицирующих записи. Приложение ссылается на запись, используя синтаксис, который виден на следующем примере путевого имени объекта:

```
\\DARYL\root\CIMV2:Win32_NTLogEvent.Logfile="Application",
                                        RecordNumber="1"
```

Первый компонент (\\DARYL) идентифицирует компьютер, на котором расположен объект, а второй компонент (\root\CIMV2) идентифицирует пространство имен, в котором этот объект находится. Имя класса следует за двоеточием, и имена ключей и связанные с ними значения следуют за точкой. Значения ключей разделяются запятой.

WMI предоставляет интерфейсы, позволяющие приложениям нумеровать все объекты конкретного класса или составлять запросы, возвращающие экземпляры классов, соответствующие критериям запроса.

Связи классов

Многие типы объектов имеют те или иные связи. Например, у объекта «компьютер» есть процессор, программное обеспечение, операционная система, активные процессы и т. д. WMI позволяет поставщикам создавать ассоциативный класс (association class) для представления логической связи между двумя разными классами. Ассоциативные классы связывают один класс с другим, поэтому они имеют только два свойства: имя класса и модификатор Ref. В следующем выводе данных показана связь, в которой MOF-файл поставщика Event Log связывает класс Win32_NTLogEvent с классом Win32_ComputerSystem. Получая объект, приложение управления может запрашивать связанные с ним объекты. Таким способом поставщик определяет иерархию объектов.

```
[dynamic: ToInstance, provider("MS_NT_EVENTLOG_PROVIDER"): ToInstance, EnumPrivileg
es{"SecurityPrivilege"}:
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. 4.20 показан WMI Object Browser (еще одно инструментальное средство из состава WMI Administrative Tools), отображающий содержимое пространства имен CIMV2. Системные компоненты Windows обычно помещают свои объекты в пространство имен CIMV2. Object Browser сначала находит местоположение экземпляра Win32_ComputerSystem-объекта ALEX-LAPTOP, которым является компьютер. Затем Object Browser получает объекты, связанные

с Win32_ComputerSystem, и показывает их ниже ALEX-LAPTOP. Пользовательский интерфейс Object Browser выводит связи объектов в виде значка папки с двойной стрелкой. Под значком папки отображаются объекты связанного типа класса.

В Object Browser можно увидеть, что ассоциативный класс поставщика Event Log Win32_NTLogEventComputer находится под ALEX-LAPTOP и что существует множество экземпляров класса Win32_NTLogEvent. Чтобы проверить, что в MOF-файле класс Win32_NTLogEventComputer определяется для связи класса Win32_ComputerSystem с классом Win32_NTLogEvent, обратитесь к предыдущему выводу данных. Выбор экземпляра Win32_NTLogEvent в Object Browser показывает свойства этого класса во вкладке Properties (Свойства) на правой панели. Microsoft создала Object Browser для помощи WMI-разработчикам в изучении их объектов, но приложения управления будут выполнять те же операции и показывать свойства или собранную информацию более доходчиво.

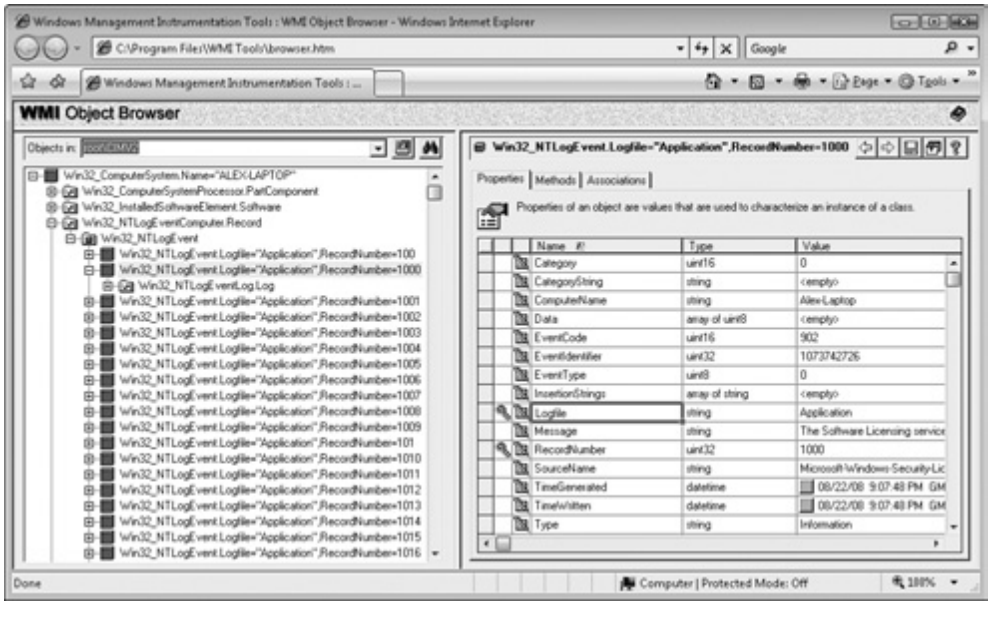


Рис. 4.20. WMI Object Browser

ЭКСПЕРИМЕНТ: ИСПОЛЬЗОВАНИЕ СЦЕНАРИЕВ WMI ДЛЯ УПРАВЛЕНИЯ СИСТЕМАМИ

Сильной стороной WMI является поддержка языков сценариев. В компании Microsoft созданы сотни сценариев, выполняющих общие административные задачи для управления учетными записями пользователей, файлами, реестром, процессами и оборудованием. Центральным хранилищем сценариев Microsoft служит веб-сайт Microsoft TechNet Scripting Center. Для использования сценария из этого центра нужно просто скопировать его текст из своего интернет-браузера, сохранить его в виде файла с расширением .vbs и запустить его с помощью команды cscript *script*.vbs, где *script* — имя, которое дано сценарию. Cscript является интерфейсом командной строки к Windows Script Host (WSH).

В качестве примера приведем сценарий TechNet, регистрирующийся на получение события создания экземпляра объекта Win32_Process, которое происходит при запуске процесса, и выводящий строку с именем процесса, который представляет объект:

```
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
Do While i = 0
    Set objLatestProcess = colMonitoredProcesses.NextEvent
    Wscript.Echo objLatestProcess.TargetInstance.Name
Loop
```

В строке, вызывающей ExecNotificationQuery, используется параметр, включающий инструкцию «select». Эта инструкция выводит на первый план поддержку со стороны WMI, предназначенного только для чтения поднабора соответствующего стандарту ANSI структурированного языка запросов — Structured Query Language (SQL), известного как WQL. Этот поднабор предназначен для предоставления потребителям WMI гибкого способа указания информации, которую им нужно извлечь из поставщиков WMI. Запуск сценария, приведенного в качестве примера с помощью Cscript, а затем запуск программы Блокнот (Notepad) приводит к следующему выводу на экран:

```
C:\>cscript monproc.vbs
Microsoft (R) Windows Script Host Version 5.7
Copyright (C) Microsoft Corporation. All rights reserved.

NOTEPAD.EXE
```

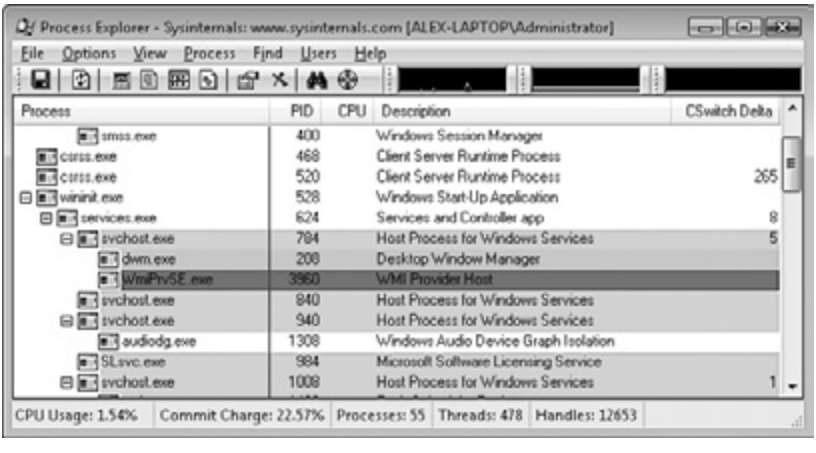
Реализация WMI

Службы WMI запускаются в общем Svchost-процессе, выполняемом под учетной записью локальной системы. Поставщики загружаются в хост-процесс поставщиков Wmiprvse.exe, который запускается в качестве дочернего по отношению к процессу службы RPC. WMI выполняет Wmiprvse под учетной записью локальной системы, локальной службы или сетевой службы, в зависимости от значения свойства HostingModel экземпляра объекта WMI Win32Provider, представляющего реализацию поставщика. Выход из процесса Wmiprvse осуществляется после того, как поставщик удаляется из кэша, через одну минуту после последнего запроса, полученного поставщиком.

Большинство компонентов WMI, включая Windows MOF-файлы, DLL-библиотеки встроенных поставщиков и WMI DLL-библиотеки приложений управления, по умолчанию находятся в каталоге %SystemRoot%\System32 и в каталоге %SystemRoot%\System32\Wbem. Загляните в каталог %SystemRoot%\System32\Wbem, и вы найдете там файл Ntevt.mof, MOF-файл поставщика Event Log. Вы также найдете там Ntevt.dll, DLL-библиотеку поставщика Event Log, используемую службой WMI.

ЭКСПЕРИМЕНТ: ПРОСМОТР СОЗДАНИЯ WMIPRVSE

Создание Wmiprvse можно увидеть, запустив Process Explorer и выполнив команду Wmic. Процесс Wmiprvse появится ниже процесса Svchost, являющегося хост-процессом RPC-службы. Если в Process Explorer включено выделение заданий (job highlighting), этот процесс появится с цветом, выделяющим задание. Для недопущения потребления всех ресурсов виртуальной памяти, имеющихся в системе, потерявшим управление поставщиком, Wmiprvse выполняется в объекте задания, ограничивающем количество создаваемых им дочерних процессов и объем виртуальной памяти каждого процесса и всех процессов, которые могут назначаться заданием (см. главу 5).



В каталогах, расположенных ниже %SystemRoot%\System32\Wbem, хранится репозиторий, файлы журналов и MOF-файлы сторонних производителей. WMI реализует репозиторий, называемый репозиторием объектов CIMOM, используя собственную версию механизма базы данных Microsoft JET. Файл базы данных по умолчанию находятся в каталоге %SystemRoot%\System32\Wbem\Repository\.

WMI следует многочисленным настройкам реестра, хранящимся в разделе реестра службы HKLM\SOFTWARE\Microsoft\WBEM\CIMOM, например пороговым величинам и максимальным значениям конкретных параметров.

Драйверы устройств используют специальные интерфейсы для предоставления данных WMI и получения от него команд. Эти интерфейсы называются системными управляющими командами WMI — WMI System Control commands — и являются частью WDM. Поскольку интерфейсы являются кроссплатформенными, они попадают в пространство имен \root\WMI.

WMIC

В состав Windows также включена утилита Wmic.exe, позволяющая взаимодействовать с WMI из оболочки командной строки, осведомленной о существовании WMI. Через эту оболочку доступны все WMI-объекты и их свойства, включая методы, что превращает WMIC в развитую консоль управления системой.

Безопасность WMI

В WMI безопасность реализуется на уровне пространства имен. Если приложение управления успешно подключилось к пространству имен, оно может просматривать свойства всех объектов в этом пространстве имен и иметь к ним доступ. Администратор может использовать приложение Управляющий элемент WMI (WMI Control) для управления тем, какие пользователи могут получать доступ к пространству имен. Внутри системы модель безопасности реализована с помощью ACL-списков и дескрипторов безопасности, являющихся частью стандартной модели безопасности Windows, реализующей проверку доступа (см. главу 6).

Для запуска приложения Управляющий элемент WMI выберите из меню Пуск (Start) пункт Панель управления (Control Panel). В панели управления выберите пункты Система и безопасность (System And Maintenance) ▸ Администрирование (Administrative Tools) ▸ Управление компьютером (Computer Management). Затем откройте ветвь Службы и приложения (Services And Applications). Щелкните правой кнопкой мыши на пункте Управляющий элемент WMI (WMI Control) и выберите пункт Свойства (Properties), чтобы запустить диалоговое окно программы Управляющий элемент WMI, показанный на рис. 4.21. Для настройки безопасности щелкните на вкладке Безопасность (Security), выберите пространство имен и щелкните на кнопке Безопасность (Security). В других вкладках диалогового окна Свойства: Управляющий элемент WMI (WMI Control Properties) можно изменить настройки производительности и создания резервных копий, в которых хранится реестр.

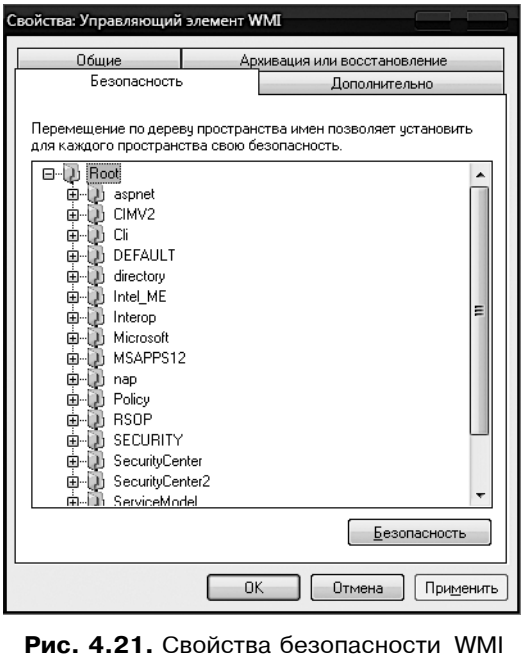


Рис. 4.21. Свойства безопасности WMI

Инфраструктура диагностики Windows

Инфраструктура диагностики Windows — Windows Diagnostic Infrastructure (WDI) — помогает обнаружить, распознать и разрешить общие проблемные ситуации с минимальным вмешательством со стороны пользователя. Компонентами Windows реализуются триггеры, заставляющие WDI запускать соответствующие ситуации модули устранения неполадок для обнаружения обстоятельств возникновения проблемы. Триггер может показать, что система близка к проблемной ситуации или уже вошла в нее. Как только модуль устранения неполадок определит причину, он может вызвать программу, предназначенную для решения проблемы. Решение может быть простым и заключаться в смене настроек реестра или во взаимодействии с пользователем для выполнения действий восстановительного или настроечного характера. В конечном счете, основная роль WDI заключается в предоставлении универсальной среды для компонентов Windows с целью выполнения задач, задействованных в автоматизированном обнаружении проблемы, установке и устранения причин ее возникновения.

Инструментарий WDI

Windows или компоненты приложения должны добавить инструментарий для уведомления WDI при возникновении проблемной ситуации. Компоненты могут ожидать результатов диагностики в синхронном режиме или могут продолжать работу и позволять диагностике проводиться в асинхронном режиме. Для поддержки этих моделей в WDI реализуется два разных типа API-функций инструментария:

- Диагностика, основанная на событиях, которая может использоваться для минимально навязчивого оснащения диагностическим инструментарием и может добавляться к компоненту, не требуя внесения каких-либо изменений в его реализацию. WDI поддерживает два вида диагностики, основанной на событиях: простые сценарии и старт-стоп сценарии. В простом сценарии ответственность за сбой и инициирование события для запуска диагностики несет одна точка в коде программы. В старт-стоп сценарии считается подверженным риску сбоя весь путь кода, и весь он оснащается для проведения диагностики. Одно событие инициируется в начале сценария для сеанса трассировки событий реального времени — real-time Event Tracing for Windows (ETW), — названного DiagLog. В то же самое время средство ядра под названием Scenario Event Mapper (SEM) включает коллекцию дополнительных средств ETW-отслеживания в средства регистрации WDI-окружения. Второе событие инициируется, чтобы дать сигнал на завершение сценария диагностики, к тому времени SEM выключает подробное отслеживание. Этот механизм «отслеживания строго по потребности» снижает издержки на подробное отслеживание, наряду с этим поддерживая сбор достаточного объема информации об окружении для WDI, чтобы найти основную причину без воспроизведения проблемы, если произойдет сбой.
- Диагностика по потребности, позволяющая приложениям самостоятельно запрашивать диагностику, взаимодействовать со средствами проведения

диагностики, получать уведомления о завершении диагностики и изменять поведение на основе результатов проведения диагностики. Инструментарий по потребности особенно полезен, когда диагностику нужно провести в привилегированном контексте безопасности. WDI способствует переносу контекста через границы доверия и процесса, а также при необходимости поддерживает заимствование прав вызывающей стороны.

Служба политики диагностики

Служба политики диагностики — Diagnostic Policy Service (DPS, %SystemRoot%\System32\Dps.dll) — реализует основную часть внутреннего интерфейса WDI-сценария. DPS является многопоточной службой (запускаемой в Svchost), которая принимает запросы сценария по потребности, а также отслеживает события диагностики, доставляемые через DiagLog. Взаимоотношения DPS с другими ключевыми компонентами WDI показаны на рис. 4.22. В ответ на эти запросы DPS запускает соответствующие модули устранения неполадок, в которых закодированы предметно-ориентированные знания, например, о том, как найти основную причину сетевых проблем. Кроме того, DPS открывает модулям доступ ко всей контекстной информации, имеющей отношение к сценарию, в форме зафиксированных результатов отслеживания. Модули устранения неполадок выполняют автоматизированный анализ данных и могут запросить у DPS запуск вторичного модуля, называемого программой решения проблемы, отвечающей за ее устранение по возможности без каких-либо видимых проявлений.

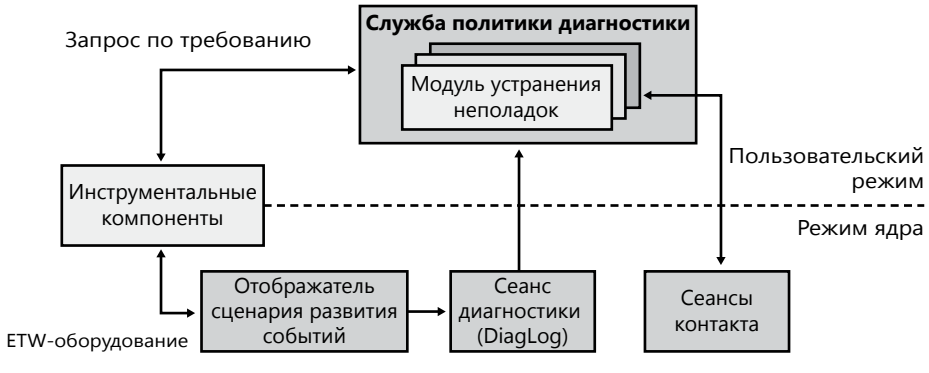


Рис. 4.22. Архитектура инфраструктуры диагностики Windows — Windows Diagnostic Infrastructure

DPS управляет настройками групповой политики (Group Policy) и навязывает ее выполнение сценариям проведения диагностики. Для настройки вариантов проведения диагностики и автоматического восстановления можно воспользоваться редактором групповой политики (%SystemRoot%\System32\Gpedit.msc). Получить доступ к этим настройкам можно, как показано на рис. 4.23, перейдя по пунктам Конфигурация компьютера (Computer Configuration) ▸ Административные шаблоны (Administrative Templates) ▸ Система (System) ▸ Диагностика (Troubleshooting And Diagnostics).

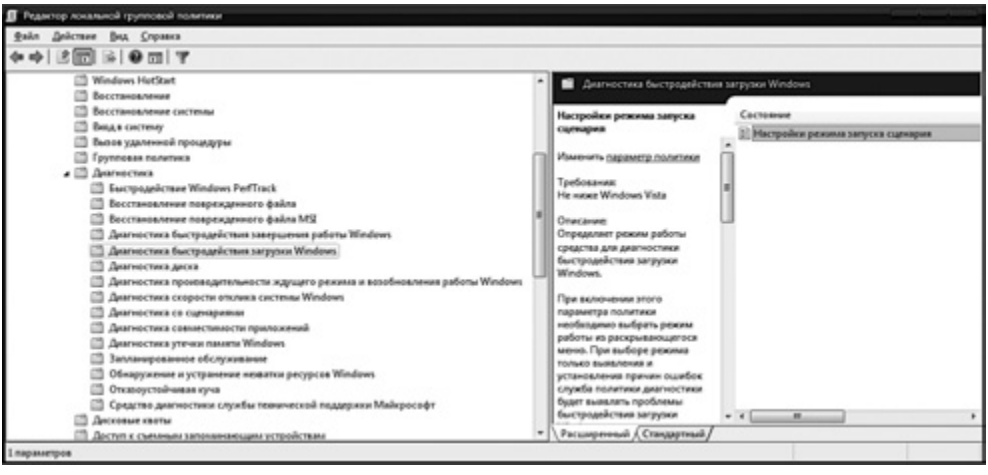


Рис. 4.23. Настройки конфигурации службы политики диагностики

Проведение диагностики

В Windows реализованы несколько встроенных сценариев проведения диагностики и предназначенных для этого утилит. Некоторые из примеров включают в себя следующие реализации:

- Диагностику дисков, включающую наличие кода технологии Self-Monitoring Analysis and Reporting Technology (SMART) внутри драйвера класса хранения (%SystemRoot%\System32\Driver\Classspnp.sys) для отслеживания здоровья диска. WDI уведомляет пользователя и проводит его через создание резервной копии данных после обнаружения предстоящего сбоя диска. Кроме этого, Windows отслеживает аварии приложений, вызванные повреждениями диска в местах записи важных системных файлов. Диагностика использует механизм защиты файлов Windows — Windows File Protection — для автоматического восстановления таких поврежденных системных файлов из резервного кэша, когда это возможно.
- Диагностику и устранение неполадок в работе сети, расширяющую WDI для обработки разных классов проблем, связанных с сетью, например с совместным использованием файлов, с доступом к Интернету, с беспроводными сетями, с брандмауэрами сторонних производителей и с общей возможностью подключения к сети. Дополнительные сведения о работе в сети даны в главе 7 «Сети».
- Предотвращение исчерпания ресурсов, куда включена диагностика утечек памяти, а также выявление и устранение исчерпания ресурсов Windows. Эта диагностика может обнаружить, когда выделенный лимит приближается к своему максимуму, и оповестить пользователей о возникшей ситуации, включая главных потребителей памяти и других ресурсов. После этого пользователь может остановить свой выбор на завершении работы таких приложений, чтобы попытаться освободить часть ресурсов.

- Средство диагностики памяти Windows, которое может быть задействовано пользователем вручную из диспетчера загрузки (Boot Manager) в ходе запуска системы или автоматически, рекомендовано к запуску системой оповещения об ошибках — Windows Error Reporting (WER) — после сбоя системы, который был проанализирован как потенциальный результат отказа оперативной памяти.
- Средство восстановления при загрузке — Windows startup repair tool, — которое пытается устранить в автоматическом режиме определенные классы ошибок, зачастую перекладывая на себя ответственность за пользователей, не способных запустить систему. К числу таких ошибок относятся неверные настройки данных конфигурации загрузки — Boot Configuration Data (BCD), — поврежденные структуры диска, такие как главная загрузочная запись — MBR — или загрузочный сектор, и поврежденные драйверы. Когда загрузка системы терпит неудачу, диспетчер загрузки автоматически запускает средство восстановления при загрузке, если оно установлено, которое также включает в себя варианты восстановления ручным способом и доступ к командной строке.
- Диагностику производительности Windows, куда включены диагностика производительности загрузки Windows, диагностика производительности завершения работы Windows, диагностика производительности приостановки и возобновления работы Windows и диагностика производительности сохранения активного состояния Windows. На основе определенных пороговых показателей расчета времени и внутреннего поведения, ожидаемого от этих механизмов, Windows может выявить проблемы, приводящие к снижению производительности, и зарегистрировать их в журнале событий, который, в свою очередь, использует WDI для предоставления разрешений и критического анализа пользователям, чтобы те попытались устранить проблему.
- Помощника по совместимости программ — Program Compatibility Assistant (PCA), — позволяющего устаревшим приложениям выполняться на последних версиях Windows, пренебрегая проблемами совместимости. PCA выявляет сбои при установке приложений, связанные с несоответствиями при проверке версии, и сбои в процессе выполнения, вызванные нерекомендуемыми двоичными кодами и настройками системы управления учетными записями пользователей — User Account Control (UAC). PCA пытается восстановить работу после этих сбоев, применяя для приложений соответствующие настройки совместимости, которые возымеют эффект при следующем запуске. Кроме этого, PCA ведет базу данных программ с известными проблемами совместимости и информирует пользователей о потенциальных проблемах, возникающих при запуске таких программ.

Заключение

Итак, мы изучили общую структуру Windows, основные системные механизмы, на которых построена эта структура, и основные механизмы управления. Заложив эту основу, мы готовы к более подробному исследованию отдельных компонентов исполняющей системы, начиная с процессов и потоков.

Глава 5. Процессы, потоки и задания

В этой главе будет рассмотрена структура данных и алгоритмы, применяемые в отношении процессов, потоков и заданий в операционной системе Microsoft Windows. В первом разделе основное внимание будет уделено внутренним структурам, составляющим процесс. Во втором разделе будут в общих чертах рассмотрены действия, предпринимаемые для создания процесса (и его исходного потока). Затем будет рассмотрено внутреннее устройство потоков и планирования их работы, и завершится глава описанием заданий.

Поскольку процессы и потоки в Windows имеют отношение к большому количеству компонентов, ряд терминов и структур данных (таких как рабочие наборы, объекты и дескрипторы, кучи системной памяти и т. д.) в этой главе будут просто упомянуты, а их подробное рассмотрение последует в остальных главах книги. Чтобы полностью разобраться с материалом данной главы, нужно ознакомиться с понятиями и концепциями, рассмотренными в главе 1 «Общие представления и инструментальные средства» и в главе 2 «Архитектура системы». Необходимо, например, понимать разницу между процессом и потоком, между пользовательским режимом и режимом ядра, а также понимать, что такое структура виртуального адресного пространства Windows.

Внутреннее устройство процессов

В этом разделе рассматриваются основные структуры данных процесса Windows, поддерживаемые различными частями системы, и различные способы и инструменты для изучения этих данных.

Структуры данных

Каждый процесс в Windows представлен структурой процесса, создаваемой в исполняющей системе (EPROCESS). Кроме множества атрибутов, относящихся к процессу, EPROCESS содержит ряд других связанных с ним структур данных и указывает на такие структуры. Например, у каждого процесса есть один или несколько потоков, каждый из которых представлен структурой потока (см. раздел «Внутреннее устройство потоков»), создаваемой в исполняющей системе (ETHREAD).

EPROCESS-структура и основная часть связанных с ней структур данных находятся в адресном пространстве системы. Единственным исключением является блок переменных окружения процесса — process environment block (PEB), — который находится в адресном пространстве процесса (в нем содержится информация, доступная коду, выполняемому в режиме пользователя). Кроме того, некоторые из структур данных процесса, используемые в управлении памятью, например список рабочего набора, действуют только в контексте текущего процесса, поскольку они хранятся в системном пространстве, определенном для процесса.

Для каждого процесса, выполняющего программу Win32, процесс подсистемы Win32 (Csrss) поддерживает параллельную структуру под названием

CSR_PROCESS. И наконец, та часть подсистемы Win32, которая работает в режиме ядра (Win32k.sys), поддерживает структуру данных для каждого процесса — W32PROCESS. Эта структура создается при первом же вызове потоком Windows-функции USER или GDI, реализованной в режиме ядра.

За исключением структуры процесса простоя (idle process), каждая структура EPROCESS инкапсулируется в виде объекта процесса с помощью диспетчера объектов исполняющей системы (см. главу 3). Поскольку процессы не являются именованными объектами, в таком инструментальном средстве, как WinObj, они не видны. Но в каталоге \ObjectTypes можно увидеть тип объекта под названием «Process». Дескриптор процесса за счет использования связанных с процессом API-функций предоставляет доступ к некоторым данным в структуре EPROCESS, а также в некоторых связанных с ней структурах.

На рис. 5.1 показана упрощенная схема структур данных процесса и потока. Каждая структура данных, отображенная на этом рисунке, будет подробно рассмотрена в данной главе.

Рис. 5.1. Структуры данных, связанные с процессами и потоками

Путем регистрации уведомлений о создании процесса многие другие драйверы и компоненты системы могут создать свои собственные структуры данных для отслеживания информации, сохраняемой ими для каждого процесса. Когда заходит разговор об издержках процесса, зачастую нужно принимать во внимание объем таких структур данных, хотя получить его точное значение почти невозможно.

Сначала давайте уделим внимание объекту процесса (объект потока будет рассмотрен чуть позже в разделе «Внутреннее устройство потоков»). Основные поля структуры EPROCESS показаны на рис. 5.2.

Структуры данных, создаваемые для процесса, выстроены по замыслу, схожему с тем, который использовался для API-функций ядра и его компонентов. (Эти

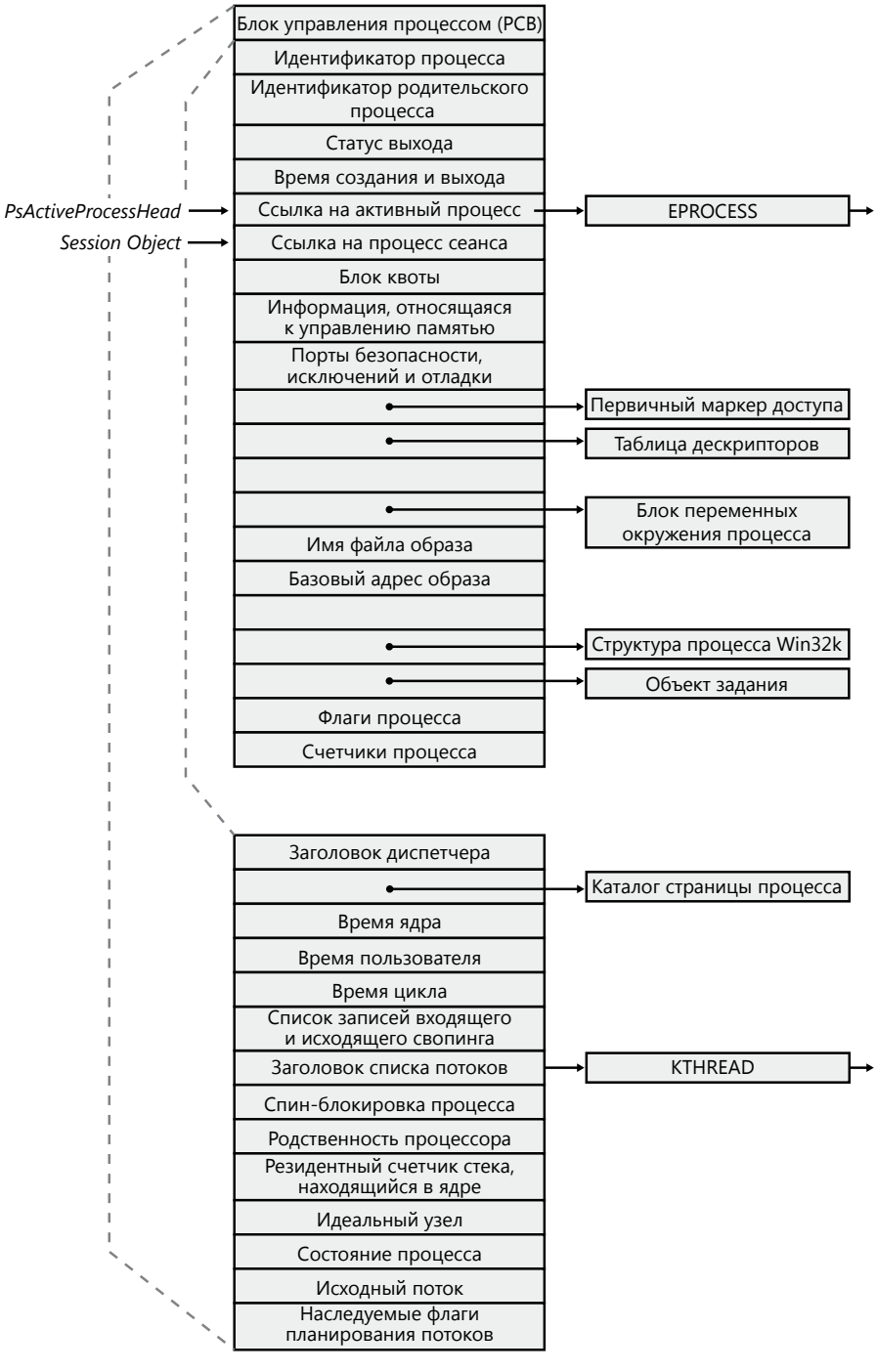


Рис. 5.2. Основные поля структуры процесса, создаваемые в исполняющей системе, и его встроенной структуры процесса, находящейся в ядре

компоненты делятся на изолированные и распределенные по уровням модули со своими собственными соглашениями об именах.)

Как показано на рис. 5.2, первый элемент структуры процесса, находящейся в исполняющей системе, называется блоком управления процессом (process control block, PCB). Эта структура относится к типу KPROCESS и находится в ядре. Хотя процедуры исполняющей системы хранят информацию в EPROCESS, код диспетчера, планировщика, а также код учета прерываний и времени, будучи частью ядра операционной системы, используют вместо нее структуру KPROCESS. Тем самым допускается существование уровня абстракции между высокоуровневой функциональностью исполняющей системы и ее базовыми низкоуровневыми реализациями конкретных функций, что помогает предотвратить нежелательные зависимости уровней друг от друга.

ЭКСПЕРИМЕНТ: ВЫВОД ФОРМАТА СТРУКТУРЫ EPROCESS И ЕЕ ПОЛЕЙ

Для вывода списка полей, составляющих структуру EPROCESS и их шестнадцатеричного смещения, наберите в отладчике ядра (см. главу 1) команду dt nt!_eprocess. Вывод (усеченный для экономии пространства) выглядит на 32-разрядной системе следующим образом:

```
lkd> dt nt!_eprocess
   +0x000 Pcb               : _KPROCESS
   +0x080 ProcessLock       : _EX_PUSH_LOCK
   +0x088 CreateTime        : _LARGE_INTEGER
   +0x090 ExitTime          : _LARGE_INTEGER
   +0x098 RundownProtect    : _EX_RUNDOWN_REF
   +0x09c UniqueProcessId   : Ptr32 Void
...
   +0x0dc ObjectTable       : Ptr32 _HANDLE_TABLE
   +0x0e0 Token             : _EX_FAST_REF
...
   +0x108 Win32Process      : Ptr32 Void
   +0x10c Job               : Ptr32 _EJOB
...
   +0x2a8 TimerResolutionLink : _LIST_ENTRY
   +0x2b0 RequestedTimerResolution : Uint4B
   +0x2b4 ActiveThreadsHighWatermark : Uint4B
   +0x2b8 SmallestTimerResolution : Uint4B
   +0x2bc TimerResolutionStackRecord : Ptr32 _PO_DIAG_STACK_RECORD
```

Первый элемент этой структуры (Pcb) является встроенной структурой типа KPROCESS. В ней хранятся данные о планировании и об учете времени. Формат структуры процесса, находящейся в ядре, можно вывести тем же способом, что и EPROCESS:

```
lkd> dt _kprocess
nt!_KPROCESS
   +0x000 Header             : _DISPATCHER_HEADER
   +0x010 ProfileListHead    : _LIST_ENTRY
   +0x018 DirectoryTableBase : Uint4B
   ...
   +0x074 StackCount         : _KSTACK_COUNT
```

продолжение ⇩

```
+0x078 ProcessListEntry : _LIST_ENTRY
+0x080 CycleTime        : Uint8B
+0x088 KernelTime       : Uint4B
+0x08c UserTime         : Uint4B
+0x090 VdmTrapcHandler  : Ptr32 Void
```

Команда dt также позволяет вам просматривать конкретное содержимое одного поля или нескольких полей путем набора их имен после имени структуры, как, например, в команде dt nt!_eprocess UniqueProcessId, которая выведет поле идентификатора процесса (process ID). Что касается поля, представляющего структуру, например поля Pcb структуры EPROCESS, которое содержит подструктуру KPROCESS, добавление точки после имени поля заставит отладчик показать подструктуру.

Например, еще один способ посмотреть на KPROCESS заключается в наборе команды dt nt!_eprocess Pcb. Можно продолжать рекурсивно пользоваться данным способом, добавляя дополнительные имена полей (с KPROCESS) и т. д. И наконец, для рекурсивного просмотра всех подструктур можно в команде dt воспользоваться ключом –r, который как раз для этого и предназначен. Добавляя число после ключа, можно управлять глубиной рекурсии, выполняемой командой.

Ранее показанная команда dt выведет формат выбранной структуры, а не содержимое какого-то конкретного экземпляра этого типа структуры. Чтобы вывести экземпляр действующего процесса, нужно в качестве аргумента указать команде dt адрес структуры EPROCESS. Адрес почти всех структур EPROCESS, имеющихся в системе, можно получить с помощью команды !process 0 0 (исключение составляет процесс простоя системы). Поскольку структура KPROCESS является первым компонентом структуры EPROCESS, адрес EPROCESS при его использовании с командой dt _kprocess будет также работать и в качестве адреса KPROCESS. ■

Процессы и потоки являются неотъемлемой частью Windows, поэтому говорить о них, не ссылаясь на многие другие части системы, просто невозможно. Но чтобы не допустить спонтанного роста размеров данной главы, темы, связанные с другими частями — управление памятью, безопасность, объекты и дескрипторы, — рассматриваются в других местах.

Блок PEB находится в адресном пространстве пользовательского режима того процесса, который он описывает. В нем содержится информация, необходимая загрузчику образа, диспетчеру кучи и другим компонентам Windows, которым необходим к нему доступ из пользовательского режима. Структуры EPROCESS и KPROCESS доступны только из режима ядра.

Основные поля блока PEB показаны на рис. 5.3 и более подробно рассмотрены позже в данной главе.

Структура CSR_PROCESS содержит информацию о процессах, характерных для подсистемы Windows (Csrss). По существу, только связанные с Windows приложения имеют связанную с ними структуру CSR_PROCESS (например, Smss ее не имеет). Кроме того, поскольку каждый сеанс имеет свой собственный экземпляр подсистемы Windows, структура CSR_PROCESS поддерживается процессом Csrss внутри каждого отдельного сеанса. Основные поля структуры CSR_PROCESS показаны на рис. 5.4 и более подробно рассмотрены в данной главе чуть позже.

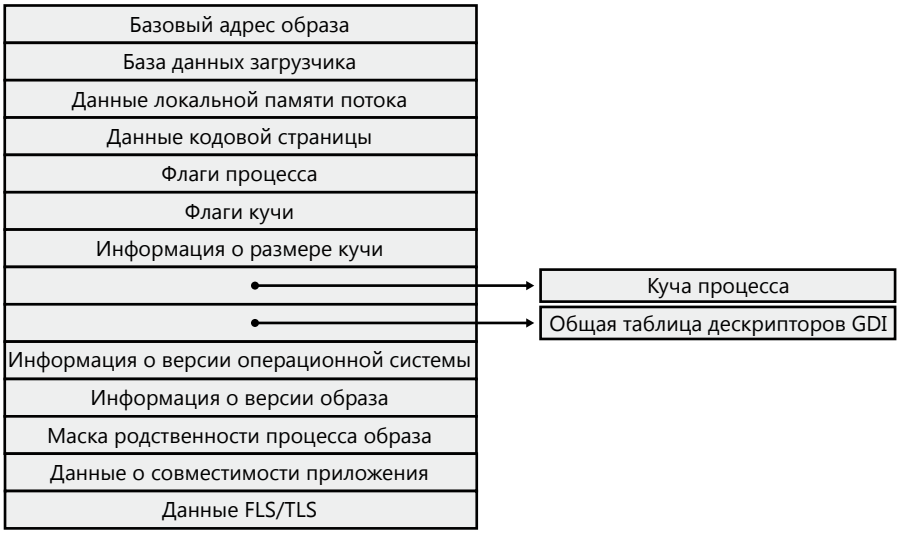


Рис. 5.3. Поля блока переменных окружения процесса

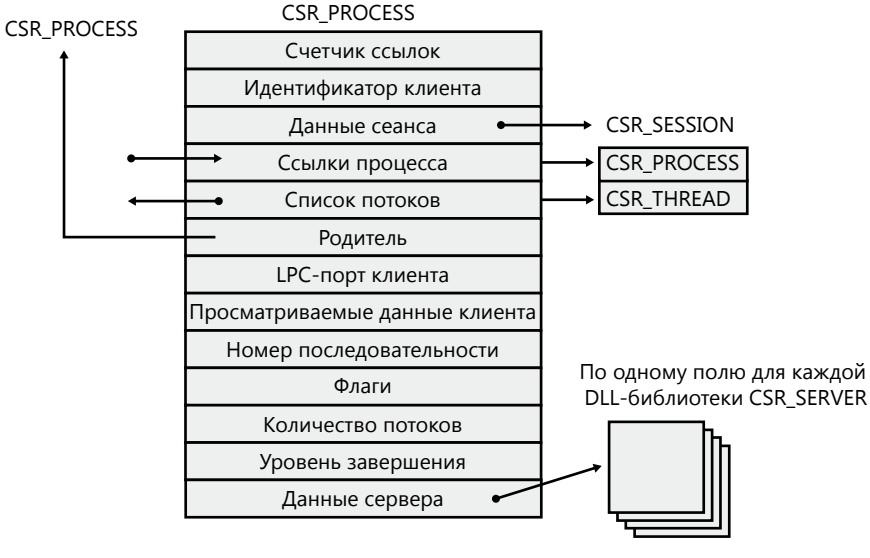


Рис. 5.4. Поля структуры CSR-процесса

ЭКСПЕРИМЕНТ: ИСПОЛЬЗОВАНИЕ КОМАНДЫ ОТЛАДЧИКА ЯДРА !PROCESS

Команда отладчика ядра !process выводит поднабор информации об объекте процесса и связанных с ним структурах. Этот вывод для каждого процесса размещается в двух частях. В первой из них можно увидеть показанную здесь информацию о процессе. (Когда не указывается адрес процесса или его идентификатор, команда !process выводит информацию о процессе,

владеющем потоком, запущенным в данный момент на центральном процессоре 0, которым на однопроцессорной системе будет сам отладчик WinDbg.)

```
lkd> !process
PROCESS 85857160  SessionId: 1  Cid: 0bcc    Peb: 7ffd9000  ParentCid: 090c
    DirBase: b45b0820  ObjectTable: b94ffda0  HandleCount:  99.
    Image: windbg.exe
    VadRoot 85a1c8e8 Vads 97 Clone 0 Private 5919. Modified 153. Locked 1.
    DeviceMap 9d32ee50
    Token                             ebaa1938
    ...
     '    PageFaultCount               37066
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      6242
```

После вывода основных сведений о процессе следует список потоков этого процесса. Эта информация рассматривается чуть позже во врезке «Эксперимент: использование команды отладчика ядра !thread» (с. 429).

К числу других команд, отображающих информацию о процессе, относится команда !handle, которая выводит дамп таблицы дескрипторов процесса (см. главу 3, раздел «Дескрипторы объекта и таблица дескрипторов процесса»). Структуры безопасности процессов и потоков рассмотрены в главе 6 «Безопасность».

Обратите внимание на то, что в выводимой информации дается адрес PEB, который можно использовать с командой !peb, показанной в следующем эксперименте по просмотру блока PEB произвольного процесса. Но поскольку PEB находится в адресном пространстве пользовательского режима, он действителен только в контексте своего собственного процесса. Чтобы посмотреть на PEB другого процесса, сначала нужно переключить WinDbg на этот процесс. Это можно сделать с помощью команды .process, сопровождаемой указателем EPROCESS. ■

ЭКСПЕРИМЕНТ: ИССЛЕДОВАНИЕ PEB

Дамп структуры блока PEB можно вывести в отладчике ядра с помощью команды !peb, которая показывает блок PEB того процесса, который владеет текущим потоком, запущенным на центральном процессоре 0. Используя информацию, полученную из предыдущего эксперимента, можно в качестве аргумента предоставить этой команде PEB-указатель.

```
lkd> !peb 7ffd9000
PEB at 7ffd9000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            No
    ImageBaseAddress:         002a0000
    Ldr                       77895d00
...
    WindowTitle:  'C:\Users\Alex Ionescu\Desktop\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
    Environment:  001850a8
        ALLUSERSPROFILE=C:\ProgramData
        APPDATA=C:\Users\Alex Ionescu\AppData\Roaming
    ...
```

ЭКСПЕРИМЕНТ: ИССЛЕДОВАНИЕ СТРУКТУРЫ CSR_PROCESS

Вывести дамп структуры CSR_PROCESS можно с помощью команды !dp в отладчике, запущенном в пользовательском режиме после подключения к Csrss-процессу сеанса, подлежащего изучению. Чтобы получить список процессов текущего сеанса, выберите пункты File (Файл), Attach To A Process (Подключиться к процессу). Вы сможете увидеть процессы сеанса, раскрыв соответствующий элемент дерева. Не забудьте установить флажок Noninvasive (Неинвазивный режим), чтобы избежать замедления работы своей системы.

В качестве входных данных команда !dp получает PID процесса, для которого выводится дамп структуры CSR_PROCESS. В качестве аргумента может быть непосредственно передан указатель структуры. Поскольку команда !dp уже выполняет внутри себя команду dt, вводить ее не нужно.

```
0:000> !dp v 0x1c0aa8-8
PCSR_PROCESS @ 001c0aa0:
   +0x000 ClientId         : _CLIENT_ID
   +0x008 ListLink         : _LIST_ENTRY [ 0x1d8618 - 0x1b1b10 ]
   +0x010 ThreadList       : _LIST_ENTRY [ 0x1c0b80 - 0x1c7638 ]
   +0x018 NtSession        : 0x001c0bb8 _CSR_NT_SESSION
...
   +0x054 Luid             : _LUID
   +0x05c ServerDllPerProcessData : [1] (null)
Threads:
Thread 001c0b78, Process 001c0aa0, ClientId 198.19c, Flags 0, Ref Count 1
Thread 001c0e78, Process 001c0aa0, ClientId 198.1cc, Flags 0, Ref Count 1
...
```

Структура W32PROCESS является заключительной структурой данных системы, которая связана с рассматриваемыми процессами. В ней содержится вся информация, необходимая коду графики Windows и коду управления окном. Этот код находится в ядре (Win32k) для поддержки информации о состоянии GUI-процессов (которые ранее были определены как процессы, которые выполнили хотя бы один системный вызов USER/GDI). Основные поля структуры W32PROCESS показаны на рис. 5.5 и более подробно будут рассмотрены в этой главе чуть позже.

ЭКСПЕРИМЕНТ: ИССЛЕДОВАНИЕ W32PROCESS

Расширениями отладчика никакой команды для вывода дампа структуры W32PROCESS не предусмотрено, но она есть в символах драйвера Win32k. Поэтому при использовании команды dt с соответствующим именем символа win32k!_W32PROCESS можно вывести дамп полей, если известен

указатель. Поскольку команда !process этот указатель не выводит (даже при том, что он хранится в объекте EPROCESS), поле должно быть изучено вручную с помощью команды dt nt!_EPROCESS Win32Process, за которой следует указатель EPROCESS.

В следующем примере показана структура W32PROCESS для оболочки Explorer.exe:

```
lkd> dt win32k!_W32PROCESS 0xff991490
   +0x000 Process             : 0x84a2b030 _EPROCESS
   +0x004 RefCount            : 1
...
   +0x020 W32Pid              : 0x590
   +0x024 GDIHandleCount      : 383
   +0x028 GDIHandleCountPeak  : 0x239
   +0x02c UserHandleCount     : 228
   +0x030 UserHandleCountPeak : 0x16c
...
   +0x088 hSecureGdiSharedHandleTable : 0x84a24159
   +0x08c DxProcess                   : 0xa2c93980
```

Поле DxProcess является указателем на еще одну структуру данных, имеющуюся у процесса, в данном случае обслуживаемую драйвером порта видеокарты DirectX, но ее описание выходит за рамки тематики данной книги.

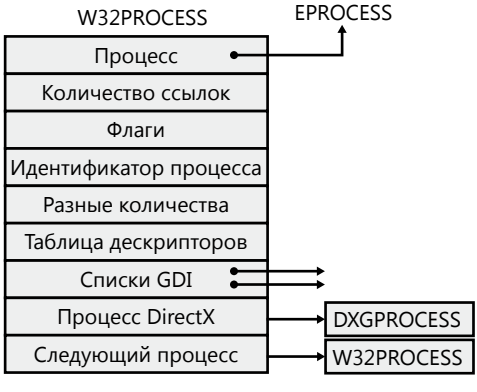


Рис. 5.5. Поля структуры процесса Win32k

Защищенные процессы

В модели безопасности Windows любой процесс, запущенный с маркером доступа, содержащим привилегию отладки (такую как у учетной записи администратора), может запросить любые требуемые ему права на доступ к любому другому процессу, запущенному на машине. Например, он может осуществлять произвольное чтение и запись в произвольных участках памяти процесса, вставлять код, приостанавливать и возобновлять выполнение потоков и запрашивать информацию о других процессах. Такие инструментальные средства, как Process Explorer и Диспетчер задач, нуждаются в этих правах и запрашивают их для предоставления пользователям своих функциональных возможностей.

Логическое поведение (помогающее гарантировать, что полный доступ к запущенному коду будет только у администраторов) входит в противоречие с поведением системы в соответствии с требованиями по управлению цифровыми правами, установленными медиа-индустрией для операционных систем компьютеров, которые требуются для поддержки проигрывания усовершенствованного цифрового контента, например, записанного на носителях Blu-ray и DVD. Для поддержки надежного и защищенного проигрывания такого контента Windows использует защищенные процессы. Такие процессы сосуществуют с обычными процессами Windows, но они накладывают существенные ограничения на права доступа, которые могут быть запрошены другими процессами системы (даже запущенными с привилегиями администратора).

Защищенные процессы могут быть созданы любым приложением, но операционная система разрешит процессу быть защищенным, только если файл образа был снабжен цифровой подписью со специальным сертификатом Windows Media Certificate. В Windows защищенный путь к носителю — Protected Media Path (PMP) — использует защищенные процессы, чтобы предоставить защиту для дорогостоящих носителей, и разработчики таких приложений, как DVD-плееры, могут воспользоваться защищенными процессами путем использования API-функций Media Foundation.

Процесс Audio Device Graph (Audiodg.exe) является защищенным, потому что с его помощью может быть декодирован защищенный музыкальный контент. Аналогично этому, принадлежащий системе отчета об ошибках Windows Error Reporting (WER) клиентский процесс (Werfault.exe) также должен быть запущен в защищенном режиме, поскольку ему нужно иметь доступ к защищенным процессам на случай аварии одного из них. И наконец, сам процесс System защищен, потому что часть расшифрованной информации генерируется драйвером Ksecdd.sys и сохраняется в его памяти пользовательского режима. Процесс System также защищен, чтобы остались в неприкосновенности все дескрипторы ядра (потому что таблица дескрипторов процесса System содержит все имеющиеся в системе дескрипторы ядра).

Поддержка защищенных процессов на уровне ядра ведется по двум направлениям: во-первых, во избежание инъекционных атак основная часть создания процесса происходит в режиме ядра. Во-вторых, у защищенных процессов имеется специальный бит, установленный в их структуре EPROCESS, который изменяет поведение процедур, связанных с мерами безопасности в диспетчере процессов для отказа в конкретных правах доступа, которые обычно предоставляются администраторам. Фактически, в отношении защищенных процессов предоставляются только следующие права: на запрос процесса и установку ограниченной информации — PROCESS_QUERY/SET_LIMITED_INFORMATION, на завершение процесса — PROCESS_TERMINATE и на приостановку и возобновление процесса — PROCESS_SUSPEND_RESUME. Определенные права доступа отключаются также и в отношении потоков, запущенных внутри защищенных процессов, эти права доступа будут рассмотрены позже в разделе «Внутреннее устройство потоков».

Поскольку в Process Explorer для запроса информации о внутренних данных процесса используются стандартные API-функции пользовательского режима, проводить определенные операции над защищенными процессами невозможно. С другой стороны, такое инструментальное средство, как WinDbg, в режиме отладки ядра, использующее для получения этой информации инфраструктуру

режима ядра, будет в состоянии вывести полную информацию. Поведение Process Explorer при работе с таким защищенным процессом, как Audiodg.exe, показано в эксперименте раздела «Внутреннее устройство потоков».

ПРИМЕЧАНИЕ

Как уже упоминалось в главе 1, для выполнения локальной отладки ядра нужно загрузить систему в режиме отладки (который включается с помощью команды bcdedit /debug on или путем использования дополнительных настроек загрузки в средстве Msconfig). Это оградит от атак на защищенные процессы и на Protected Media Path (PMP), рассчитанных на использование режима отладки. При загрузке в режиме отладки проигрывание содержимого высокого разрешения работать не будет.

Надежное ограничение прав доступа позволяет ядру оградить защищенный процесс от доступа к нему из пользовательского режима. С другой стороны, поскольку признаком защищенного процесса служит флаг в структуре EPROCESS, администратор все же может загрузить драйвер режима ядра, снимающий бит этого флага. Но это будет нарушением PMP-модели и будет считаться злонамеренным действием, и такой драйвер, скорее всего, будет в конечном итоге заблокирован от загрузки на 64-разрядной системе, поскольку действующая в режиме ядра политика цифровой подписи кода запретит ставить подпись на вредоносный код. Даже на 32-разрядных системах драйвер должен быть распознан PMP-политикой, иначе воспроизведение будет остановлено. Эта политика реализована компанией Microsoft и обнаруживается не на всяком ядре. Блокировка потребует вмешательства Microsoft по идентификации подписи, определения вредоносности кода и обновления ядра.

Порядок работы функции CreateProcess

Итак, в этой главе уже были показаны различные структуры данных, используемые при работе с состоянием процесса и при управлении процессом, а также возможность исследования этой информации с помощью различных инструментальных средств и команд отладчика. В данном разделе мы увидим, как и когда эти структуры данных создаются и заполняются, а также получим общее представление о том, как осуществляется создание процесса и завершение его работы.

Процесс подсистемы Windows создается, когда приложение вызывает одну из функций создания процесса или каким-то образом попадает в нее. К таким функциям относятся CreateProcess, CreateProcessAsUser, CreateProcessWithTokenW и CreateProcessWithLogonW. Создание процесса Windows состоит из нескольких этапов, выполняемых тремя частями операционной системы: Windows-библиотекой Kernel32.dll, работающей на стороне клиента (при выполнении процедур CreateProcessAsUser, CreateProcessWithTokenW и CreateProcessWithLogonW часть этой работы сначала делается библиотекой Advapi32.dll), исполняющей системой Windows и процессом подсистемы Windows (Csrss).

Поскольку архитектура подсистем Windows имеет множество сред окружения, создание объекта процесса исполняющей системы (который может использо-

ваться другими подсистемами) отделено от работы, выполняемой при создании процесса подсистемы Windows. Поэтому, хотя следующее описание порядка выполнения Windows-функции CreateProcess будет довольно сложным для восприятия, нужно иметь в виду, что часть работы характерна для семантики, добавленной подсистемой Windows, в отличие от основной работы, необходимой для создания объекта процесса исполняющей системы.

В следующем списке собраны основные этапы создания процесса с помощью Windows-функции CreateProcess. Операции, выполняемые на каждом этапе, подробно рассматриваются в следующих разделах. Некоторые из этих операций должны быть выполнены самой функцией CreateProcess (или другими вспомогательными процедурами в пользовательском режиме), а другие операции будут выполнены функцией NtCreateUserProcess или одной из ее вспомогательных процедур в режиме ядра. В нашем следующем подробном анализе будет обозначена разница между двумя этими типами операций, необходимыми на каждом этапе.

Основные этапы создания процесса с помощью Windows-функции CreateProcess:

1. Проверка приемлемости параметров; преобразование флагов и настроек подсистемы Windows в их более подходящие аналоги; анализ, проверка приемлемости и преобразование списка атрибутов в его более подходящий аналог.
2. Открытие файла образа (.exe), предназначенного для выполнения внутри процесса.
3. Создание объекта процесса исполняющей системы.
4. Создание исходного потока (стека, контекста и объекта потока исполняющей системы Windows).
5. Выполнение инициализации характерного для подсистемы Windows процесса, запускаемого после создания данного процесса.
6. Начало выполнения исходного потока (если только не был установлен флаг создания приостановленного потока — CREATE_SUSPENDED).
7. Завершение в контексте нового процесса и потока инициализации адресного пространства (например, загрузка требуемых DLL-библиотек) и начало выполнения программы.

На рис. 5.6 показан обзор этапов, которым следует Windows при создании процесса.

Этап 1. Преобразование и проверка приемлемости параметров и флагов

Перед тем как открыть на запуск исполняемый образ, функция CreateProcess выполняет действия, описанные ниже.

В CreateProcess для нового процесса определяется класс приоритета в виде независимых битов в параметре CreationFlags. Поэтому для одного вызова функции CreateProcess можно указать более одного класса приоритетов. Windows решает вопрос назначения процессу класса приоритета путем выбора для него набора самого низкого класса приоритета.

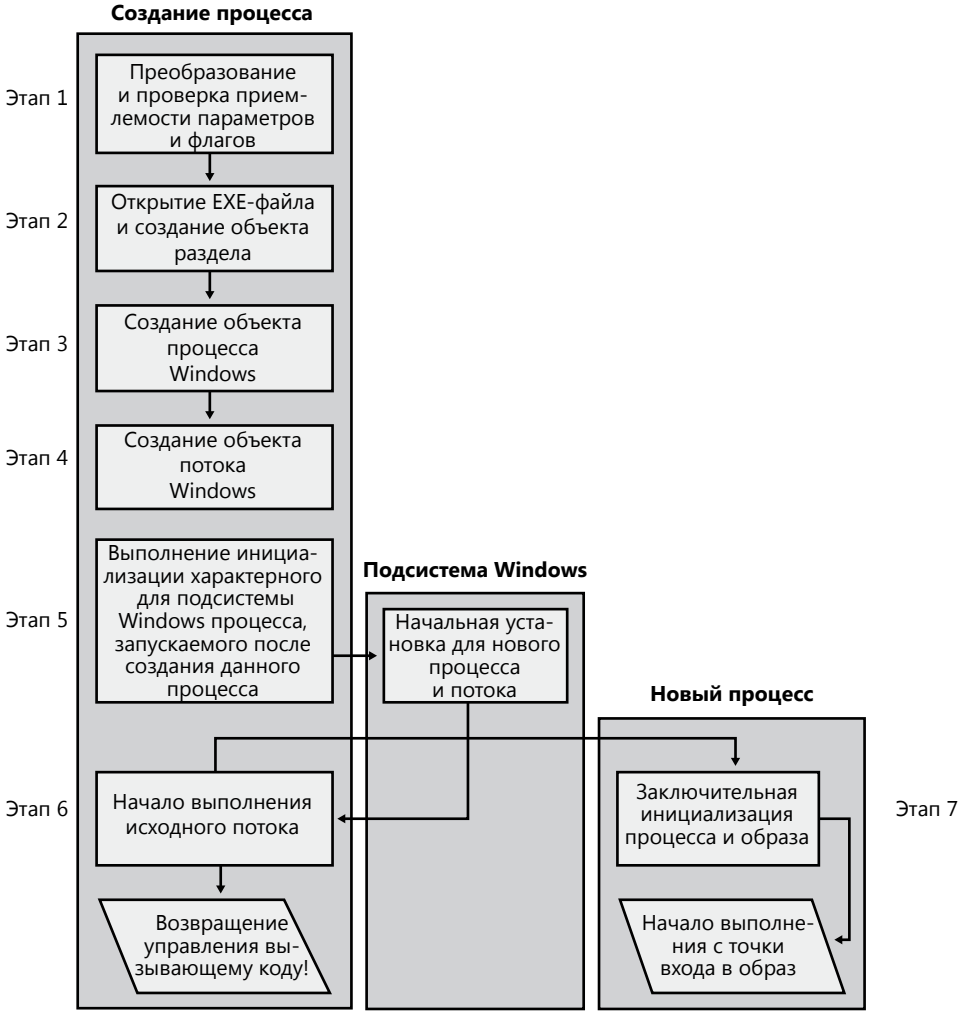


Рис. 5.6. Основные этапы создания процесса

Если класс приоритета для нового процесса не указан, по умолчанию для него устанавливается класс приоритета обычный — Normal, если только класс приоритета процесса, который его создал, не был Idle (Простой) или Below Normal (ниже обычного), в таком случае класс приоритета нового процесса будет таким же, как и у его создателя.

Если для нового процесса указан класс приоритета Real-time (выполнения в режиме реального времени) и код, вызвавший процесс, не имеет привилегии на повышение приоритета при планировании — Increase Scheduling Priority, — вместо него используется класс приоритета High (высокий). Иными словами, функция CreateProcess не дает сбой только потому, что у вызывающего кода были недостаточные привилегии для создания процесса с классом приоритета Real-time; просто у нового процесса не будет такого высокого приоритета, как Real-time.

Все окна связаны с рабочими столами, графическим отображением рабочего пространства. Если в вызове функции CreateProcess рабочий стол не указан, процесс связывается с текущим рабочим столом вызывающего кода.

Если процесс является частью объекта задания, но флаги создания требуют отдельной виртуальной DOS-машины — virtual DOS machine (VDM), флаг такого требования игнорируется.

Если вызывающий код отправляет монитору дескриптор, являющийся дескриптором вывода, а не дескриптором консоли, стандартные флаги дескриптора игнорируются.

Если флаги создания определяют, что процесс будет подвергаться отладке, Kernel32 приступает к подключению к исходному коду отладки в Ntdll.dll путем вызова функции DbgUiConnectToDbg и получает дескриптор объекта отладки из блока переменных окружения (TEB) текущего потока.

Библиотека Kernel32.dll устанавливает по умолчанию жесткий режим ошибки, если таковой указан во флагах создания.

Указанный пользователем список атрибутов преобразуется из формата подсистемы Windows в более подходящий формат и к нему добавляются внутренние атрибуты. Возможные атрибуты, которые могут быть добавлены к списку атрибутов, перечислены в табл. 5.1, куда включены и их документированные аналоги из Windows API, если таковые имеются.

ПРИМЕЧАНИЕ

Список атрибутов, переданный вызову функции CreateProcess, допускает обратную передачу вызывающему коду информации, выходящей за рамки простого кода статуса, например адрес TEB исходного потока или сведения о разделе образа. Это необходимо при использовании защищенных процессов, поскольку родитель не может запросить эту информацию после создания дочернего процесса.

Таблица 5.1. Обработка атрибутов

Исходный атрибут	Эквивалентный атрибут Windows	Тип	Описание
PS_CP_PARENT_PROCESS	PROC_THREAD_ATTRIBUTE_PARENT_PROCESS. Также используется при подъеме	Ввод	Дескриптор родительского процесса
PS_CP_DEBUG_OBJECT	Отсутствует — применяется при использовании флага DEBUG_PROCESS	Ввод	Объект отладки, если началась отладка процесса
PS_CP_PRIMARY_TOKEN	Отсутствует — применяется при использовании CreateProcessAsUser/WithToken	Ввод	Маркер доступа процесса, если была использована процедура CreateProcessAsUser
PS_CP_CLIENT_ID	Отсутствует — возвращается в виде параметра Win32 API	Вывод	Возвращение TID и PID исходного потока и процесса

продолжение ⇩

Таблица 5.1 *(продолжение)*

Исходный атрибут	Эквивалентный атрибут Windows	Тип	Описание
PS_CP_TEB_ADDRESS	Отсутствует — используется внутри системы и не выводится на экран	Вывод	Возвращение адреса TEB исходного потока
PS_CP_FILENAME	Отсутствует — используется в качестве параметра в API-функции CreateProcess	Ввод	Имя создаваемого процесса
PS_CP_IMAGE_INFO	Отсутствует — используется внутри системы и не выводится на экран	Вывод	Возвращает SECTION_IMAGE_INFORMATION, где содержится информация о версии, флагах и подсистеме исполняющей системы, а также размер стека и точка входа
PS_CP_MEM_RESERVE	Отсутствует — используется внутри системы процессами SMSS и CSRSS	Ввод	Массив фиксированного распределения виртуальной памяти, который должен быть создан в ходе создания начального адресного пространства процесса, предоставляя гарантированную доступность, поскольку другого распределения пока не было
PS_CP_PRIORITY_CLASS	Отсутствует — передается в виде параметра API-функции CreateProcess	Ввод	Класс приоритета, который будет дан процессу
PS_CP_ERROR_MODE	Отсутствует — передается через флаг CREATE_DEFAULT_ERROR_MODE	Ввод	Жесткий режим обработки ошибок для процесса
PS_CP_STD_HANDLE_INFO	PROC_THREAD_ATTRIBUTE_HANDLE_LIST	Ввод	Определяется, если стандартные дескрипторы должны быть продублированы или если должны быть созданы новые дескрипторы
PS_CP_HANDLE_LIST		Ввод	Список дескрипторов, принадлежащих родительскому процессу, который должен быть унаследован новым процессом

Исходный атрибут	Эквивалентный атрибут Windows	Тип	Описание
PS_CP_GROUP_ AFFINITY	PROC_THREAD_ATTRIBUTE_ GROUP_AFFINITY	Ввод	Группа (группы) процессоров, на которой потоку разрешено запускаться
PS_CP_PREFERRED_ NODE	PROC_THREAD_ ATTRIBUTES_PRFERRED_ NODE	Ввод	Предпочитаемый (идеальный) узел, который должен быть связан с процессом. Касается узла, на котором будет создана куча исходного процесса и стек потока
PS_CP_IDEAL_ PROCESSOR	PROC_THREAD_ ATTTRIBUTE_IDEAL_ PROCESSOR	Ввод	Предпочитаемый (идеальный) процессор, на котором должно планироваться выполнение потока
PS_CP_UMS_THREAD	PROC_THREAD_ATTRIBUTE_ UMS_THREAD	Ввод	Содержит UMS-атрибуты, список завершения и контекст
PS_CP_EXECUTE_ OPTIONS	PROC_THREAD_ MITIGATION_POLICY	Ввод	Содержит информацию, на основе которой для процесса могут разрешаться и запрещаться миграции (SEHOP, ATL-эмуляция, NX)

Как только эти действия завершатся, функция CreateProcess выполняет начальный вызов функции NtCreateUserProcess для попытки создания процесса. Поскольку к этому моменту времени Kernel32.dll еще не знает, является ли названный образ приложения Windows-приложением или приложением POSIX, 16-разрядным или DOS-приложением, вызов может окончиться неудачей, в таком случае функция CreateProcess ищет причину ошибки и пытается выправить ситуацию.

Этап 2. Открытие образа, предназначенного для выполнения

Как показано на рис. 5.7, первым действием функция NtCreateUserProcess ищет соответствующий Windows-образ, который запустит исполняемый файл, указанный вызывающему коду, и создаст объект раздела, чтобы затем отобразить его на адресное пространство нового процесса. Если по какой-нибудь причине вызов потерпит неудачу, управление вернется функции CreateProcess со статусом сбоя (см. табл. 5.2), что заставит CreateProcess повторить попытку выполнения.

Если указанный исполняемый файл является Windows-файлом с расширением .exe, функция NtCreateUserProcess пытается открыть файл и создать для него объект раздела. Объект пока не отображается на память, но он открывается. Успешное открытие объекта раздела еще не означает, что файл действительно

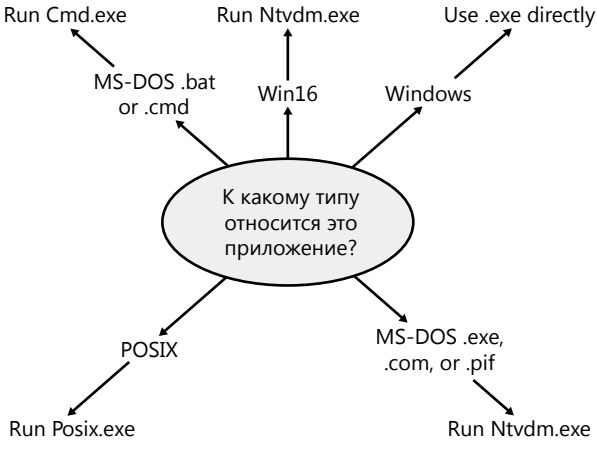


Рис. 5.7. Выбор активируемого Windows-образа

является Windows-образом, это может быть DLL-библиотека или исполняемый POSIX-файл. Если это исполняемый POSIX-файл, образ, который должен быть запущен, изменяется на Posix.exe, и функция CreateProcess перезапускается с самого начала этапа 1. Если файл является DLL-библиотекой, функция CreateProcess дает сбой.

После того как функция NtCreateUserProcess нашла настоящий исполняемый Windows-образ, она обращается в раздел реестра HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options в поисках подраздела с именем файла и расширением исполняемого образа (но без каталога и информации о пути, например *Image*.exe). Если такой подраздел существует, функция PspAllocateProcess ищет в этом подразделе параметр Debugger. Если такой параметр существует, образ, предназначенный для запуска, становится строкой в этом параметре, и функция CreateProcess перезапускается с этапа 1.

СОВЕТ

Вы можете воспользоваться таким поведением при создании процесса и отладить пусковой код процессов служб Windows до их запуска вместо подключения отладчика после запуска службы, не позволяющего отладить пусковой код.

С другой стороны, если образ не является Windows-файлом с расширением .exe (например, если это MS-DOS-, Win16- или POSIX-приложение), функция CreateProcess проходит через серию действий по поиску поддерживающего Windows-образа, чтобы его запустить. Этот процесс необходим, потому что приложения, не являющиеся Windows-приложениями, не запускаются напрямую — вместо этого Windows использует один из нескольких специальных поддерживающих образов, которые, в свою очередь, отвечают за реальный запуск программ, не являющихся Windows-программами. Например, при попытке запустить POSIX-приложение функция CreateProcess распознает его как приложение данного типа и меняет запускаемый образ на исполняемый Windows-файл Posix.exe. При попытке запустить исполняемый файл MS-DOS или Win16,

образом, который будет запускаться, станет исполняемый Windows-файл Ntvdm.exe. Короче говоря, вам не удастся напрямую создать процесс, не являющийся Windows-процессом. Если Windows не сможет решить, какой образ нужно активировать в качестве Windows-процесса (как показано в табл. 5.2), функция CreateProcess потерпит неудачу.

Таблица 5.2. Дерево решений для этапа 2 выполнения функции CreateProcess

Если образ...	Он создает код состояния	Этот образ запустит...	...в результате чего происходит
Исполняемый файл POSIX	PsCreateSuccess	Posix.exe	CreateProcess перезапускается с этапа 1
Приложение MS-DOS с расширением .exe, .com или .pif	PsCreateFailOnSectionCreate	Ntvdm.exe	CreateProcess перезапускается с этапа 1
Приложение Win16	PsCreateFailOnSectionCreate	Ntvdm.exe	CreateProcess перезапускается с этапа 1
Приложение Win64 на 32-разрядной системе (или двоичный файл PPC, MIPS или Alpha)	PsCreateFailMachineMismatch	Не определено	CreateProcess терпит неудачу
Имеет параметр Debugger с другим именем образа	PsCreateFailExeName	Имя указано в параметре Debugger	CreateProcess перезапускается с этапа 1
Недопустимый или поврежденный Windows EXE	PsCreateFailExeFormat	Не определено	CreateProcess терпит неудачу
Не может открыться	PsCreateFailOnFileOpen	Не определено	CreateProcess терпит неудачу
Командная процедура (приложение с расширением .bat или .cmd)	PsCreateFailOnSectionCreate	Cmd.exe	CreateProcess перезапускается с этапа 1

Более конкретно дерево решений, по которому проходит функция CreateProcess при запуске образа, выглядит следующим образом:

- Если образ является приложением MS-DOS с расширением .exe, .com или .pif, подсистеме Windows отправляется сообщение о проверке наличия созданного для этого сеанса процесса, поддерживающего MS-DOS (Ntvdm.exe, указанного в параметре реестра HKLM\SYSTEM\CurrentControlSet\Control\WOW\cmdline). Если поддерживающий процесс был создан, он используется для запуска приложения MS-DOS. (Подсистема Windows отправляет процессу VDM (виртуальной DOS-машины) сообщение о запуске нового образа.) Функция CreateProcess возвращает управление. Если поддерживающий процесс создан не был, запускаемый образ изменяется на Ntvdm.exe, и функция CreateProcess перезапускается с этапа 1.

- Если у запускаемого файла расширение .bat или .cmd, запускаемым образом становится Cmd.exe, окно командной строки Windows, и функция CreateProcess перезапускается с этапа 1. Имя пакетного файла передается Cmd.exe в качестве первого аргумента.
- Если образ является исполняемым файлом Win16 (Windows 3.1), функция CreateProcess должна решить, должен ли для его запуска быть создан новый VDM-процесс или должен использоваться исходный VDM-процесс, предназначенный для всего сеанса (который к этому моменту может быть еще не создан). Это решение управляется флагами функции CreateProcess под именами CREATE_SEPARATE_WOW_VDM и CREATE_SHARED_WOW_VDM. Если эти флаги не определены, поведение по умолчанию диктуется значением параметра HKLM\SYSTEM\CurrentControlSet\Control\WOW\DefaultSeparateVDM. Если приложение должно запускаться в отдельной машине VDM, запускаемый образ меняется на ntvdm.exe, за которым следуют некоторые настроечные параметры и имя 16-разрядного процесса, и функция CreateProcess перезапускается с этапа 1. В противном случае подсистема Windows отправляет сообщение, чтобы понять, существует ли общий VDM-процесс и может ли он быть использован. (Если VDM-процесс запущен на другом рабочем столе или он не запущен на том же уровне безопасности, что и вызывающий код, он не может быть использован, и должен быть создан новый VDM-процесс.) Если может использоваться общий VDM-процесс, подсистема Windows отправляет ему сообщение, чтобы запустить новый образ, и функция CreateProcess возвращает управление. Если VDM-процесс еще не был создан (или если он существует, но не может использоваться), запускаемый образ должен поменяться на образ поддержки VDM, и функция CreateProcess перезапускается с этапа 1.

Этап 3. Создание объекта процесса исполняющей системы Windows (PspAllocateProcess)

На данный момент функция NtCreateUserProcess открыла подходящий исполняемый файл Windows и создала объект раздела для отображения его на адресное пространство нового процесса. Затем она создает объект процесса исполняющей системы Windows для запуска образа путем вызова внутренней системной функции PspAllocateProcess. Создание объекта процесса исполняющей системы (осуществляемое путем создания потока) включает в себя следующие подэтапы:

- настройка объекта EPROCESS;
- создание исходного адресного пространства процесса;
- инициализация находящейся в ядре структуры процесса (KPROCESS);
- настройка PEB;
- завершение настройки адресного пространства процесса (что включает в себя инициализацию списка рабочего набора и дескрипторов виртуального адресного пространства, а также отображение образа на адресное пространство).

ПРИМЕЧАНИЕ

Единственный случай отсутствия родительского процесса относится к инициализации системы. После нее для предоставления контекста безопасности для нового процесса всегда требуется родительский процесс.

Этап 3А. Настройка объекта EPROCESS

Этот подэтап включает в себя следующие действия:

1. Наследование родственности родительского процесса, если только она не была конкретно установлена в ходе создания процесса (через список атрибутов).
2. Выбор идеального узла, который был указан в списке атрибутов, если он был указан.
3. Наследование приоритета ввода-вывода и выгружаемой памяти (page priority) от родительского процесса. Если родительского процесса не было, используется исходный приоритет выгружаемой памяти (5) и ввода-вывода (Normal).
4. Установка для статуса выхода из нового процесса значения STATUS_PENDING.
5. Выбор из списка атрибутов жесткого режима обработки ошибок; в противном случае наследование родительского режима обработки, если режим не был задан. Если родительского процесса не было, использование режима обработки по умолчанию, который заключается в отображении всех ошибок.
6. Сохранение идентификатора родительского процесса в поле InheritedFromUniqueProcessId объекта нового процесса.
7. Запрос раздела реестра Image File Execution Options с целью проверки, должен ли процесс отображаться с использованием больших страниц. Также запрос раздела, чтобы проверить, перечислена ли библиотека NTDLL в качестве DLL-библиотеки, которая должна быть отображена с помощью больших страниц внутри этого процесса.
8. Запрос раздела реестра Image File Execution Options с целью проверки, не было ли связано с процессом назначение конкретного NUMA-узла. Назначение может быть основано либо на наследовании (при котором NUMA-узел будет распространен от родителя), либо быть явным назначением NUMA-узла, если только это назначение не отменяет исходный NUMA-узел, указанный в списке атрибутов.
9. Выключение случайного выбора адреса стека, если в исполняемом приложении, составляющем процесс, выключено случайное расположение в адресном пространстве — Address space layout randomization (ASLR).
10. Попытка получения всех привилегий, необходимых для создания процесса. Выбор для процесса класса приоритета Real-time, присваивание новому процессу маркера доступа, отображение процесса с помощью больших страниц и создание процесса в рамках нового сеанса — все эти операции требуют соответствующих привилегий.
11. Создание для процесса первичного маркера доступа (дублирование первичного маркера доступа родительского процесса). Новые процессы наследуют профиль безопасности у своих родителей. Если для указания других маркеров доступа использовалась функция CreateProcessAsUser, то происходит соответствующее изменение маркера доступа. Это изменение может произойти только в том случае, если уровень целостности родительского маркера доступа доминирует над уровнем целостности маркера доступа и если маркер доступа является дочерним или смежным по отношению к родительскому. Следует заметить, что при наличии у родителя привилегии SeAssignPrimaryToken эти проверки будут обойдены.

12. Проверка идентификатора сеанса маркера доступа нового процесса с целью определения, не является ли это создание межсеансовым, в случае чего родительский процесс временно прикрепляется к целевому сеансу для корректной обработки квот и создания адресного пространства.
13. Установка блока квот нового процесса на адрес блока квот его родительского процесса и увеличение показания счетчика ссылок для родительского блока квот. Если процесс был создан с помощью функции CreateProcessAsUser, это действие будет пропущено. Вместо него будет создана квота по умолчанию или квота, соответствующая выбранному профилю пользователя.
14. Установка максимального и минимального значений рабочего набора в соответствии со значениями PspMinimumWorkingSet и PspMaximumWorkingSet. Эти значения могут быть заменены, если в подразделе PerfOptions раздела Image File Execution Options были указаны настройки производительности, в таком случае максимальное значение рабочего набора берется оттуда. Следует заметить, что по умолчанию для рабочего набора устанавливаются мягкие ограничения (soft limits), являющиеся по сути рекомендательными, а максимум рабочего набора в PerfOptions относится к жестким ограничениям (то есть рабочему набору не будет разрешено расти выше указанного числа).
15. Инициализация адресного пространства процесса (см. этап 3Б). Затем открепление от целевого сеанса, если от отличался от текущего.
16. Выбор для процесса групповой родственности, если не использовалось наследование этой родственности. Исходная групповая родственность либо будет унаследована от родителя, если ранее было установлено распространение NUMA-узла (будет использовано групповое владение NUMA-узлом), либо будет назначена по кругу, где в качестве исходного будет использовано значение PspProcessGroupAssignment. Если система находится в принудительном режиме осведомленности о группах и алгоритмом выбора была выбрана группа 0, вместо нее выбирается группа 1, если она существует.
17. Инициализация KPROCESS, как части объекта процесса (см. этап 3В).
18. Установка для процесса маркера доступа.
19. Установка для процесса нормального класса приоритета, если только родительским процессом не использовался класс простоя или класс ниже нормального, в таком случае наследуется класс приоритета родительского процесса. Если класс приоритета процесса был установлен не автоматически, а с помощью списка атрибутов, устанавливается именно этот класс.
20. Инициализация таблицы дескрипторов процесса. Если для родительского процесса установлен флаг наследования дескрипторов, все наследуемые дескрипторы копируются из таблицы дескрипторов родительского объекта в новый процесс (см. главу 3). Могут быть также использованы атрибуты процесса, но только лишь для указания поднабора дескрипторов, чем можно воспользоваться при использовании функции CreateProcessAsUser для ограничения тех объектов, которые должны быть унаследованы дочерним процессом.
21. Применение настроек производительности, если таковые были указаны в подразделе PerfOptions. В подраздел PerfOptions включаются заменители ограничений рабочего набора, приоритета ввода-вывода, приоритета выгру-

жаемой памяти и класса приоритета центрального процессора, применяемые для процесса.

22. Вычисление и установка окончательного класса приоритета процесса и квот по умолчанию для его потоков.
23. Завершение второго этапа установки адресного пространства, включая инициализацию PEB (см. этап 3Г/3Д).
24. Установка настроек миграции для поддержки неисполняемых модулей.
25. Установка PID процесса и времени его создания, хотя пока еще PID не вставлен в таблицу идентификаторов PID, так же как и процесс не вставлен в списки процессов (это задача для этапа вставки).

Этап 3Б. Создание исходного адресного пространства процесса

Исходное адресное пространство процесса состоит из следующих страниц:

- таблицы страниц (возможно, и не одной для систем с более чем двухуровневыми таблицами страниц, например для систем x86 в PAE-режиме или для 64-разрядных систем);
- страницы гиперпространства;
- страницы битового массива VAD;
- список рабочего набора.

Для создания этих страниц предпринимаются следующие действия:

1. Для отображения исходных страниц создаются записи в соответствующих таблицах страниц.
2. Количество страниц вычитается из значения переменной ядра MmTotalCommittedPages и добавляется к значению переменной MmProcessCommit.
3. Используемый по умолчанию общесистемный размер минимального рабочего набора процесса (PsMinimumWorkingSet) вычитается из MmResidentAvailablePages.
4. Таблица страниц разбивается на страницы для глобального системного пространства. (То есть помимо страниц, относящихся к процессу, которые только что были рассмотрены, и за исключением памяти, относящейся к конкретному сеансу.)

Этап 3В. Создание находящейся в ядре структуры процесса

Следующий этап работы функции PspAllocateProcess заключается в инициализации структуры KPROCESS (Pcb-представителя EPROCESS). Эта работа выполняется функцией KeInitializeProcess, которая инициализирует:

- Список с двойной связью, который соединяет все потоки, являющиеся частью процесса (изначально пустой).
- Начальное (или восстановленное) значение кванта времени процесса, используемое по умолчанию (более подробно рассмотренного далее в разделе «Пла-

нирование потоков»), которому жестко задается значение 6 до последующей инициализации с помощью PspComputeQuantumAndPriority.

ПРИМЕЧАНИЕ

Для клиентской и серверной систем Windows по умолчанию используются разные используемые по умолчанию начальные значения квантов времени. Для получения дополнительных сведений о квантах потоков нужно обратиться к разделу «Планирование потоков».

- Базовый приоритет процесса, устанавливаемый на основе вычислений, сделанный на этапе 3А.
- Значение групповой родственности в качестве значения родственности процессора, используемого по умолчанию для потоков процесса.
- Состояние свопинга процесса, устанавливаемое в резидентное (resident).
- Источник потока основывается на идеальном процессоре, который выбран ядром для этого процесса (этот выбор основывается на идеальном процессоре ранее созданного процесса, эффективный случайный выбор которого ведется на циклической основе). Создание нового процесса приведет к обновлению источника в KeNodeBlock (в исходном блоке NUMA-узла), чтобы следующий новый процесс получил для источника другой идеальный процессор.

Этап 3Г. Завершение настройки адресного пространства процесса

Настройка адресного пространства для нового процесса происходит непросто, поэтому давайте рассмотрим все по отдельным действиям. Чтобы извлечь из этого раздела максимум пользы, нужно иметь некоторое представление о внутреннем устройстве диспетчера памяти Windows.

1. Диспетчер виртуальной памяти устанавливает в качестве значения последнего времени подгонки процесса текущее время. Диспетчер рабочего набора (который работает в контексте диспетчера балансировки системного потока) использует это значение для определения, когда происходила точная настройка исходного рабочего набора.
2. Диспетчер памяти инициализирует список рабочего набора процесса — теперь могут приниматься ошибки обращения к страницам.
3. Раздел (созданный при открытии файла образа) теперь отображается на адресное пространство нового процесса, и базовый адрес раздела процесса устанавливается на базовый адрес образа.
4. Библиотека Ntdll.dll отображается на процесс; если это процесс Wow64, отображается также и 32-разрядная библиотека Ntdll.dll.
5. Если требуется, для процесса создается новый сеанс. Это особое действие реализовано главным образом для облегчения работы диспетчера сеанса — Session Manager (SMSS) при инициализации нового сеанса.
6. Стандартные дескрипторы дублируются, и новые значения записываются в структуру параметров процесса.

7. Обрабатываются любые фиксированные распределения памяти (memory reservations), перечисленные в списке атрибутов. Кроме того, два флага позволяют зарезервировать первый 1 Мбайт или первые 16 Мбайт адресного пространства. Эти флаги используются внутри системы для отображения, к примеру, кода постоянного запоминающего устройства и векторов реального режима (должны быть в нижних диапазонах виртуального адресного пространства, где обычно располагаются куча и другие структуры процесса).
8. Параметры пользовательского процесса записываются в процесс, копируются и исправляются (то есть переводятся из абсолютной формы в относительную, чтобы понадобился только один блок памяти).
9. Информация о родственности записывается в PEB.
10. На процесс отображается набор перенаправления MinWin API.

ПРИМЕЧАНИЕ

POSIX-процессы клонирую адресное пространство своих родительских процессов, поэтому им не нужно проходить через эти действия, чтобы создать новое адресное пространство. В случае работы с POSIX-приложениями базовый адрес раздела нового процесса устанавливается на базовый адрес его родительского процесса, и для нового процесса клонируется родительский блок PEB.

Этап 3Д. Настройка PEB

Функция NtCreateUserProcess вызывает функцию MmCreatePeb, которая сначала отображает таблицы общесистемной поддержке национальных языков — national language support (NLS) на адресное пространство процесса. Затем эта функция вызывает функцию MiCreatePebOrTeb для выделения страницы для PEB с последующей инициализацией нескольких полей, значения большинства из которых основаны на значениях внутренних переменных, которые были настроены через реестр, например через значения параметров MmHeap*, MmCriticalSectionTimeout и MmMinimumStackCommitInBytes. Некоторые из этих полей могут быть заменены настройками связанных исполняемых образов, например Windows-версией PE-заголовка или маской родственности в каталоге конфигурации загрузки PE-заголовка.

Если установлен флаг характеристики заголовка образа IMAGE_FILE_UP_SYSTEM_ONLY (указывающий на то, что образ может запускаться только на однопроцессорной системе), для запуска всех потоков нового процесса выбирается один и тот же центральный процессор (MmRotatingUniprocessorNumber). Процесс выбора выполняется простым циклическим проходом доступных процессоров, при каждом запуске образа этого типа используется следующий процессор. Таким образом образы этого типа распределяются между процессорами равномерно.

Этап 3Е. Завершение настройки объекта процесса исполняющей системы (PspInsertProcess)

Перед тем как сможет быть возвращен дескриптор нового процесса, должен быть завершен ряд финальных действий, выполняемых функцией PspInsertProcess и ее вспомогательными функциями:

1. Если включен общесистемный аудит процессов (либо в результате настроек локальной политики, либо в результате настроек групповой политики из контроллера домена), в журнале событий безопасности делается запись о создании процесса.
2. Если родительский процесс входил в задание, это задание восстанавливается из набора уровня заданий родительского процесса, а затем привязывается к сеансу вновь созданного процесса. И наконец, новый процесс добавляется к заданию.
3. Функция PspInsertProcess вставляет объект нового процесса в конец Windows-списка активных процессов (PsActiveProcessHead).
4. Принадлежащий родительскому процессу порт отладки процесса копируется в новый дочерний процесс, если, конечно, не был установлен флаг NoDebugInherit (значение которого может быть запрошено при создании процесса). Если порт отладки был указан, он в этот момент подключается к новому процессу.
5. Поскольку теперь объекты заданий могут определять ограничения, касающиеся той группы или тех групп, в которых могут запускаться потоки внутри процессов, являющихся частью задания, функция PspInsertProcess должна убедиться в том, что групповая родственность, связанная с процессом, не нарушает групповой родственности, связанной с заданием. Есть еще интересный второстепенный вопрос, требующий рассмотрения, не предоставляют ли полномочия задания доступ к изменению полномочий родственности процесса, поскольку менее привилегированный объект задания может столкнуться с требованиями родственности более привилегированного процесса.
6. И наконец, функция PspInsertProcess создает дескриптор нового процесса, вызывая функцию ObOpenObjectByPointer, а затем возвращает этот дескриптор вызывающему коду. Следует заметить, что пока внутри создаваемого процесса не будет создан первый поток, функция обратного вызова создания процесса не будет отправлена, и код всегда отправляет функции обратного вызова процесса перед отправкой функций обратного вызова, основанных на управлении объектами.

Этап 4. Создание исходного потока, а также его стека и контекста

К этому времени настройка объекта процесса исполняющей системы Windows полностью завершена. Но потоков пока еще нет, поэтому что-либо сделать не представляется возможным. Теперь настало время приступить к соответствующей работе. Обычно за все аспекты создания потока отвечает функция PspCreateThread, и при создании нового потока именно она и вызывается функцией NtCreateThread. Но поскольку исходный поток создается ядром внутри системы без ввода из пользовательского режима, вместо нее вызываются две вспомогательные процедуры, от которых зависят функции PspCreateThread: PspAllocateThread и PspInsertThread.

PspAllocateThread управляет собственно созданием и инициализацией самого объекта потока исполняющей системы, а PspInsertThread управляет созданием дескриптора потока и атрибутов безопасности, а также вызовом функции

KeStartThread для превращения объекта исполняющей системы в планируемый поток системы. Но поток пока еще ничего не будет делать, он создан в приостановленном состоянии и не возобновит выполнение, пока процесс не будет полностью проинициализирован (согласно описанию этапа 5).

ПРИМЕЧАНИЕ

Параметром потока (который не может быть указан в CreateProcess, но может быть определен в CreateThread) является адрес PEB. Этот параметр будет использоваться кодом инициализации, который запускается в контексте этого нового потока (согласно описанию этапа 6).

Функция PspAllocateThread выполняет следующие действия:

1. Не позволяет потокам, работа которых планируется в пользовательском режиме — user-mode scheduling (UMS), создаваться в процессах Wow64, а также не позволяет вызывающим процедурам пользовательского режима создавать потоки в системном процессе.
2. Создает и инициализирует объект потока исполняющей системы.
3. Если включено ограничение скорости центрального процессора, инициализирует блок квоты центрального процессора.
4. Инициализирует различные списки, используемые LPC, системой управления вводом-выводом и исполняющей системой.
5. Устанавливает для потока время создания и создает его идентификатор потока — thread ID (TID).
6. Перед тем как поток сможет выполняться, ему нужен стек и контекст, в котором он будет запущен, поэтому она настраивает и то и другое. Размер стека для исходного потока берется из образа, способа указать другой размер не существует. Если процесс относится к Wow64, будет также инициализирован контекст потока Wow64.
7. Распределяет для нового потока его блок переменных окружения — thread environment block (TEB).
8. Сохраняет стартовый адрес потока пользовательского режима в ETHREAD. Это адрес предоставленной системой функции запуска потока в библиотеке Ntdll.dll (RtlUserThreadStart). Указанный пользователем стартовый адрес Windows сохраняется в ETHREAD в другом месте, чтобы такое средство отладки, как Process Explorer, могло запросить информацию.
9. Вызывает для настройки структуры KTHREAD функцию KeInitThread. Для исходного приоритета потока и текущего базового приоритета устанавливаются значения базового приоритета процесса, а для их родственности и квант времени устанавливаются такие же значения, как и у процесса. Эта функция также устанавливает для потока исходный идеальный процессор (см. раздел «Идеальный и последний процессор»). Затем функция KeInitThread выделяет для потока стек ядра и инициализирует для потока аппаратный контекст, зависящий от машины, включая фреймы контекста, системных прерываний и исключений. Контекст потока устанавливается таким образом, чтобы поток запускался в режиме ядра в функции KiThreadStartup. И наконец, функция

KeInitThread устанавливает состояние потока в Initialized (Инициализирован) и возвращает управление функции PspAllocateThread.

10. Если это UMS-поток, вызывается функция PspUmsInitThread для инициализации UMS-состояния.

Как только эта работа завершится, функция NtCreateUserProcess вызывает функцию PspInsertThread, с помощью которой выполняются следующие действия:

1. Проводится проверка с целью убедиться в том, что групповая родственность потока не нарушает ограничений задания (которые были рассмотрены ранее). В ходе создания процесса эта проверка была пропущена, поскольку она была проведена на более раннем этапе.
2. Проводится проверка с целью убедиться в том, что процесс не был уже завершен, поток не был уже завершен или что поток даже еще не был в состоянии приступить к работе. Если имеет место любой из этих случаев, создание потока заканчивается неудачей.
3. Путем вызова функции KeStartThread инициализируется структура KTHREAD, как часть объекта потока. Здесь предполагается наследование у процесса-владельца настроек планирования, установка идеального узла и процессора, обновление групповой родственности и вставка потока в список процесса, который ведется структурой KPROCESS. Кроме того, на системах x64 еще один общесистемный список процессов, KiProcessListHead, используется функцией PatchGuard для соблюдения целостности списка активных процессов — PsActiveProcessHead, исполняющей системы. И наконец, увеличивается значения счетчика стека процесса.
4. Увеличивается значение счетчика потоков в объекте процесса и наследуются приоритет ввода-вывода процесса-владельца и приоритет выгружаемой памяти. Если достигнуто наивысшее количество когда-либо имевшихся у процесса потоков, обновляется также наивысший показатель счетчика потоков. Если поток был вторым для процесса, замораживается первичный маркер доступа (то есть он не может быть больше изменен, если только процесс не является процессом подсистемы POSIX).
5. Если поток является потоком UMS, увеличивается значение счетчика потоков UMS.
6. Поток вставляется в список потоков процесса, и поток приостанавливается, если создающий его процесс требует этого.
7. Если включено ограничение скорости центрального процессора, инициализируется APC контроля скорости, и в структуре KTHREAD устанавливается бит CpuThrottled.
8. Объект вставляется, и вызываются любые зарегистрированные функции обратного вызова. Если это был первый поток процесса (и поэтому операции проводились, как часть выполнения функции CreateProcess), вызываются также зарегистрированные функции обратного вызова процесса ядра.
9. С помощью функции ObOpenObjectByPointer создается дескриптор.
10. Поток подготавливается к выполнению с помощью вызова функции KeReadyThread. Он вносится в очередь готовых к выполнению отложенных потоков, процесс выгружается из памяти и запрашивается загрузка нужной страницы.

Этап 5. Выполнение следующих за инициализацией действий, относящихся к подсистеме Windows

После того как функция NtCreateUserProcess вернет управление с кодом успешного завершения работы, будут созданы все необходимые объекты процессов и потоков исполняющей системы. Затем для завершения инициализации процесса функции библиотеки Kernel32.dll выполняют различные операции, связанные с операциями, характерными для подсистемы Windows.

В первую очередь проводятся различные проверки относительно того, должна ли Windows позволить запуститься исполняющей системе. В их число входят проверка версии образа в заголовке и проверка, не заблокирован ли процесс сертификацией Windows-приложения (через групповую политику). В специализированных выпусках Windows Server 2008 R2, таких как Windows Web Server 2008 R2 и Windows HPC Server 2008 R2, проводятся дополнительные проверки, чтобы определить, не импортировало ли приложение какие-нибудь запрещенные API-функции.

Если в отношении программного обеспечения действуют ограничительные политики, для нового процесса создается маркер с ограничениями. После этого отправляется запрос базе данных совместимости приложения, чтобы определить, есть ли для процесса запись либо в реестре, либо в базе данных системного приложения. Но пока прокладки совместимости применяться не будут, информация будет сохранена в PEB, как только начнется выполнение исходного потока (на этапе 6).

К этому моменту Kernel32.dll отправляет сообщение подсистеме Windows, чтобы она могла настроить для нового процесса SxS-информацию, например файлы манифестов, пути перенаправлений DLL и выполнение вне процесса. Также для процесса и исходного потока инициализируются структуры подсистемы Windows. Сообщение включает в себя следующую информацию:

- дескрипторы процесса и потока;
- записи во флагах создания;
- идентификатор создателя процесса;
- флаг, показывающий, принадлежит ли процесс Windows-приложению (чтобы Csrss мог определить, нужно или нет показывать курсор запуска);
- информация о языке пользовательского интерфейса;
- DLL-перенаправление и флаги .local;
- информация файла манифеста.

При получении этого сообщения подсистема Windows выполняет следующие действия:

1. Функция CsrCreateProcess создает дубликат дескриптора для процесса и потока. Здесь же значение счетчика использования процесса и потока увеличивается со значения 1 (установленного во время создания) до значения 2.
2. Если не указан класс приоритета процесса, функция CsrCreateProcess устанавливает его в соответствии с алгоритмом, рассмотренным ранее в данном разделе.
3. Размещает структуру процесса Csrss (CSR_PROCESS).
4. Устанавливает порт исключения нового процесса в качестве общего функционального порта для подсистемы Windows, чтобы эта подсистема получала

сообщение, когда в процессе возникнет исключение второго шанса. (Более подробные сведения об обработке исключений даны в главе 3.)

5. Размещает и инициализирует структуру потока Csrss (CSR_THREAD).
6. Функция CsrCreateThread вставляет поток в список потоков процесса.
7. Увеличивает значение счетчика процессов в сеансе.
8. Устанавливает для уровня завершения процесса значение 0x280 (уровень завершения процесса по умолчанию, дополнительные сведения можно найти в документации MSDN Library, в разделе SetProcessShutdownParameters).
9. Вставляет структуру нового Csrss-процесса в список процессов, относящихся ко всей подсистеме Windows.
10. Размещает и инициализирует структуру данных, создаваемую для каждого процесса и используемую той частью подсистемы Windows, которая выполняется в режиме ядра (W32PROCESS).
11. Выводит стартовый курсор приложения. Этот курсор похож на вращающийся бублик, таким образом Windows сообщает пользователю: «Я кое-что запустила, но вы можете в данный промежуток времени пользоваться курсором». Если процесс не осуществляет после двух секунд вызова графического интерфейса пользователя — GUI, возвращает стандартную форму указателя. Если процесс вызывает GUI в предоставленное для этого время, функция CsrCreateProcess ждет пять секунд, пока приложение не покажет окно. По прошествии этого времени функция CsrCreateProcess опять восстанавливает внешний вид курсора.

После того как Csrss выполнит эти действия, функция CreateProcess проверяет, не был ли процесс запущен с повышенными правами (запущен через функцию ShellExecute и повышен в правах службой AppInfo после того, как пользователь ответил на соответствующий запрос). Сюда включена проверка, не был ли процесс программой установки. Если был, маркер процесса открыт, и флаг виртуализации установлен так, чтобы приложение было виртуализировано (см. главу 6). Если приложение содержит прокладку повышения прав или имело в своем манифесте запрос на повышение уровня прав, процесс ликвидирован, а запрос на повышение прав оправлен службе AppInfo (см. главу 6).

Следует заметить, что большинство этих проверок не проводятся для защищенных процессов, потому что эти процессы должны были предназначаться для Windows Vista или более поздних версий Windows, где не было причин для запроса повышенных прав, виртуализации или проверок совместимости приложений и соответствующей обработки. Кроме того, механизмы, допускающие выполнение, например механизмы прокладок, использующие свои обычные технологии перехвата и внесения исправлений в память, превращались бы в дыры безопасности, если кто-нибудь нашел способ вставки произвольных прокладок, изменяющих поведение защищенных процессов. В дополнение к этому, поскольку механизм прокладки (Shim Engine) устанавливается родительским процессом, у которого не должно быть доступа к его дочернему защищенному процессу, не может работать даже узаконенная технология прокладок.

Этап 6. Начало выполнения исходного потока

К этому моменту уже определено окружение процесса, выделены ресурсы для использования его потоками, у процесса есть поток, и подсистема Windows

знает о новом процессе. Если вызывающий код не установил флаг о создании приостановленного процесса — CREATE_SUSPENDED, то теперь исходный поток приведен в состояние готовности и может приступить к работе, выполнив оставшуюся часть работы по инициализации процесса, которая осуществляется в контексте нового процесса (этап 7).

Этап 7. Выполнение инициализации процесса в контексте нового процесса

Новый поток начинает свою жизнь с выполнения в режиме ядра процедуры запуска потока KiThreadStartup. Эта процедура снижает IRQL-уровень потока с уровня отложенного вызова процедуры или уровня диспетчеризации — deferred procedure call (DPC)/dispatch — до уровня APC, а затем вызывает системную процедуру исходного потока PspUserThreadStartup. Этой процедуре в качестве параметра передается определенный пользователем стартовый адрес потока.

В первую очередь эта функция отключает возможность смены первичного маркера доступа процесса, которая предназначена только для поддержки POSIX (для эмуляции поведения setuid). Затем на основе информации, имеющейся в структурах данных режима ядра, в TEB устанавливаются локальный идентификатор (Locale ID) и идеальный процессор, после чего проверяется, не произошел ли сбой при создании потока. Затем вызывается функция DbgkCreateThread, которая проверяет, отправлены ли новому процессу уведомления образа. Если они не отправлены и уведомления разрешены, то сначала уведомление образа отправляется процессу, а затем оно отправляется функции загрузки образа в Ntdll.dll. Следует заметить, что это делается на данном этапе, а не в момент первого отображения образов, потому что к тому времени идентификатор процесса (требуемый для внешних вызовов ядра) еще не был назначен.

После завершения этих проверок выполняется еще одна проверка, не подвергается ли процесс отладке. Если подвергается, функция PspUserThreadStartup проверяет, были ли отправлены уведомления об отладке этому процессу. Если не были, сообщение о создании процесса отправляется через объект отладки (если таковой имеется), чтобы событие отладки запуска процесса (CREATE_PROCESS_DEBUG_INFO) могло быть отправлено соответствующему процессу отладчика. Далее следует аналогичное событие отладки запуска потока и еще одно событие отладки для загрузки образа Ntdll.dll. Затем функция DbgkCreateThread ждет ответа от отладчика (через функцию ContinueDebugEvent).

Теперь, после того как отладчик был оповещен, функция PspUserThreadStartup смотрит на результат начальной проверки жизни потока. Если он был отменен при запуске, поток завершается. Эта проверка производится после уведомлений отладчика и образа, чтобы убедиться в том, что отладчики режима ядра и пользовательского режима не пропустили информации о потоке, даже если поток никогда не получал шанса на запуск.

В противном случае процедура проверяет, разрешена ли в системе предварительная выборка приложения, и если разрешена, вызывает механизм предварительной выборки (и Superfetch) для обработки файла инструкции предвыборки (если таковой имеется) и осуществления предварительной выборки страниц, на которые были ссылки в течение первых 10 секунд последнего запуска процесса.

Затем функция PspUserThreadStartup проверяет, был ли уже установлен общесистемный cookie-файл в структуре SharedUserData. Если он еще не был

установлен, функция генерирует его на основе хэша системной информации, такой как количество обработанных прерываний, доставок DPC и ошибок обращения к страницам. Этот общесистемный cookie-файл используется во внутренних декодированиях и кодированиях указателей, например в диспетчере кучи для защиты от определенных классов применения.

И наконец, функция PspUserThreadStartup устанавливает контекст исходного переходника (initial thunk) для запуска процедуры инициализации загрузчика образа (LdrInitializeThunk в Ntdll.dll), а также общесистемной заглушки запуска потока — thread startup stub (RtlUserThreadStart в Ntdll.dll). Эти действия совершаются путем редактирования контекста потока на месте с последующим вызовом выхода из операции системной службы, которая загружает специально созданный контекст пользователя. Процедура LdrInitializeThunk инициализирует загрузчик, диспетчер кучи, NLS-таблицы, массивы локальной памяти потока — thread-local storage (TLS) и локальной памяти волокна — fiber-local storage (FLS), а также структуры критического раздела. Затем она загружает любые требуемые библиотеки DLL и вызывает с помощью функционального кода DLL_PROCESS_ATTACH точки входа DLL.

Как только функция вернет управление, процедура NtContinue возвращает контекст нового пользователя и возвращается в пользовательский режим — вот теперь действительно запускается выполнение потока.

Функция RtlUserThreadStart использует адрес фактической точки входа образа и пусковой параметр и вызывает код точки входа приложения. Адрес и параметр уже были помещены в стек ядром. Эта довольно сложная череда событий преследует две цели. Во-первых, она позволяет загрузчику образа внутри Ntdll.dll провести настройки процесса закулисно внутри самой системы, чтобы другой код пользовательского режима мог должным образом работать. (В противном случае у него не будет кучи, локальной памяти потока и т. д.)

Во-вторых, когда все потоки начинаются в общей процедуре, это позволяет им быть заключенными в оболочку обработки исключений, чтобы при их аварии библиотека Ntdll.dll знала об этом и могла вызвать фильтр необработанного исключения внутри библиотеки Kernel32.dll. Это позволяет также скоординировать выход потока по возвращении из стартовой процедуры потока и выполнение различной работы по очистке. Разработчики приложений могут также вызвать функцию SetUnhandledExceptionFilter, чтобы добавить их собственный код обработки необработанных исключений.

ЭКСПЕРИМЕНТ: ОТСЛЕЖИВАНИЕ ЗАПУСКА ПРОЦЕССА

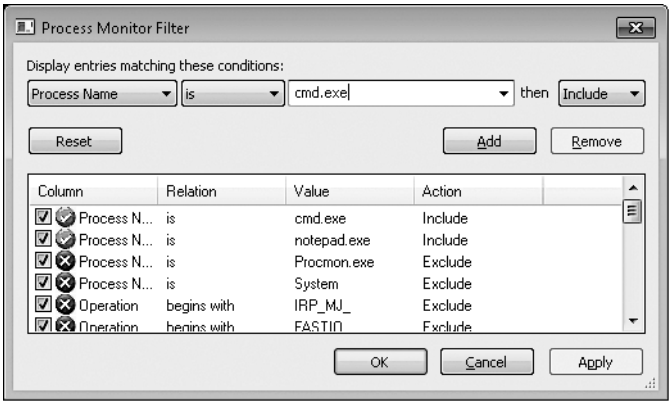
После подробного изучения порядка запуска процесса и различных операций, требуемых для начала выполнения приложения, мы собираемся воспользоваться средством Process Monitor, чтобы посмотреть на файловый ввод-вывод и разделы реестра, к которым было обращение в ходе этого процесса.

Хотя этот эксперимент не дает полной картины всех рассмотренных нами внутренних действий, вы сможете посмотреть на некоторые части системы в действии, а именно на предвыборку и Superfetch, на варианты выполнения файла-образа и на другие проверки совместимости, а также на отображение DLL-библиотеки загрузчика образа.

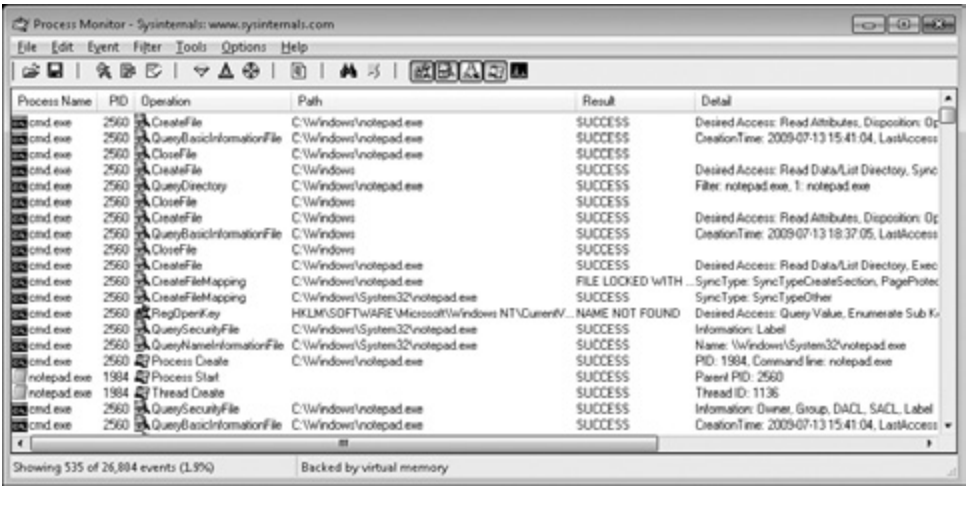
Мы посмотрим на весьма простой исполняемый файл — Notepad.exe — и запустим его из окна командной строки (Cmd.exe). Существенно то, что

мы посмотрим на операции как внутри Cmd.exe, так и внутри Notepad.exe. Вспомним, что большой объем работы в пользовательском режиме выполняется функцией CreateProcess, которая вызывается родительским процессом перед тем, как ядро создаст новый объект процесса.

Чтобы все правильно настроить, добавьте два фильтра к средству Process Monitor: один для Cmd.exe и один для Notepad.exe — нужно включить только эти два процесса. Убедитесь в том, что не имеется никаких текущих запущенных экземпляров, чтобы вы были уверены, что наблюдаете нужные события. Окно фильтров должно иметь следующий вид.



Затем убедитесь в том, что отключена регистрация событий (снят флажок захвата событий — Capture Events в меню File), и запустите окно командной строки. Включите регистрацию событий (путем повторного использования меню File или просто нажав комбинацию клавиш CTRL+E или щелкнув на значке увеличительного стекла на панели инструментов), а затем наберите Notepad.exe и нажмите клавишу Ввод. На обычной Windows-системе вы должны увидеть появление где-то между 500 и 1500 событий. Скройте столбцы Sequence (Последовательность) и Time Of Day (Время дня), чтобы сосредоточиться на интересующих нас столбцах. Ваше окно должно приобрести следующий вид.

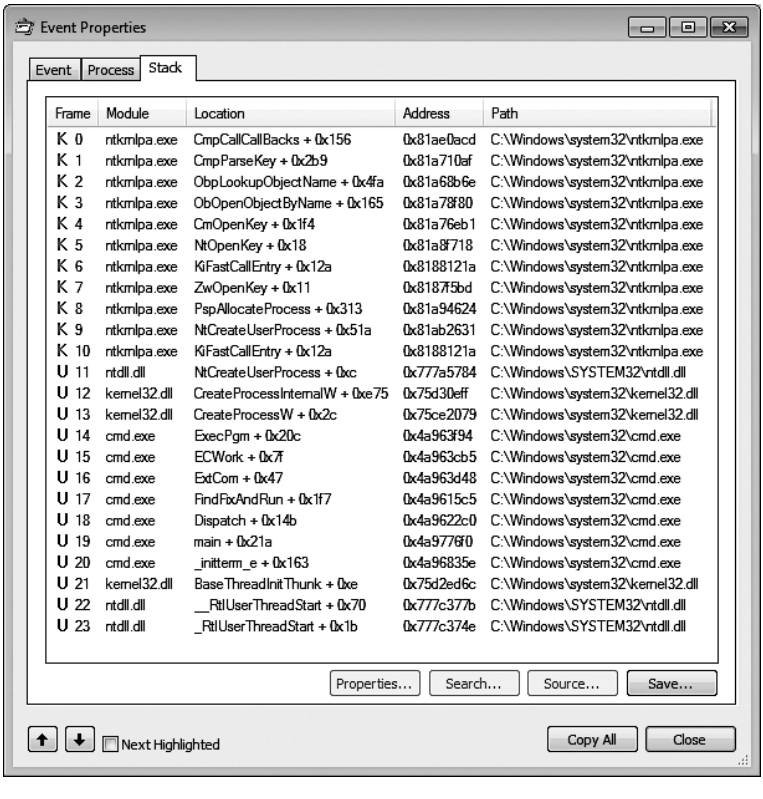


Согласно описанию этапа 1 хода выполнения функции CreateProcess, в первую очередь нужно заметить, что перед запуском процесса и созданием первого потока Cmd.exe считывает значение параметра реестра HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options. Поскольку никаких настроек на образ выполнения, связанных с Notepad.exe нет, процесс создается сам по себе.

Как в случае этого, так и в случае любого другого события, зарегистрированного в журнале Process Monitor, у вас есть возможность, изучая стек событий, посмотреть, в каком режиме, пользовательском или режиме ядра, выполнялась каждая часть создания процесса и с помощью каких процедур. Для этого дважды щелкните на событии RegOpenKey и перейдите во вкладку Stack (Стек). На следующей копии экрана показан стандартный стек на 32-разрядной Windows-машине.

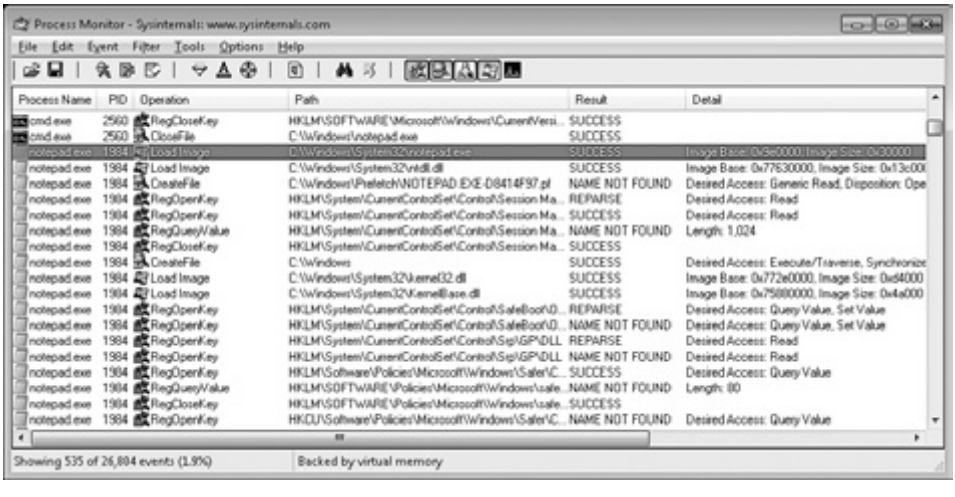
Этот стек показывает, что вы уже достигли той части создания процесса, которая выполняется в режиме ядра (посредством функции NtCreateUserProcess), и что за данную проверку отвечает вспомогательная процедура PspAllocateProcess.

Спускаясь по списку событий после того, как были созданы потоки и процессы, вы заметите три группы событий. Первая является простой проверкой флагов совместимости приложения, которые должны позволить коду создания процесса, выполняемому в пользовательском режиме, знать, нужно ли через механизм прокладки проводить проверки внутри базы данных совместимости приложения.

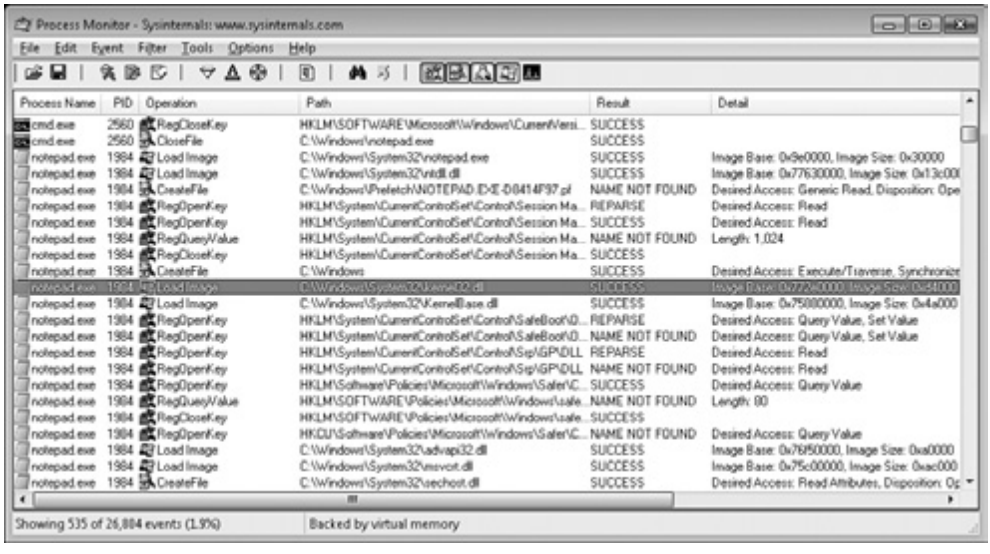


За этой проверкой следуют несколько считываний разделов Side-By-Side, Manifest и MUI/Language, являющихся частью упомянутой ранее структуры сборки. И наконец, можно увидеть файловый ввод-вывод в отношении одного или нескольких файлов с расширением .sdb, которые представляют в системе базы данных совместимости приложений. Этот ввод-вывод осуществляется с целью дополнительных проверок, не нужно ли для этого приложения вызывать какой-нибудь механизм прокладки. Поскольку Блокнот (Notepad) является программой Microsoft с обычным поведением, ему не нужны никакие прокладки.

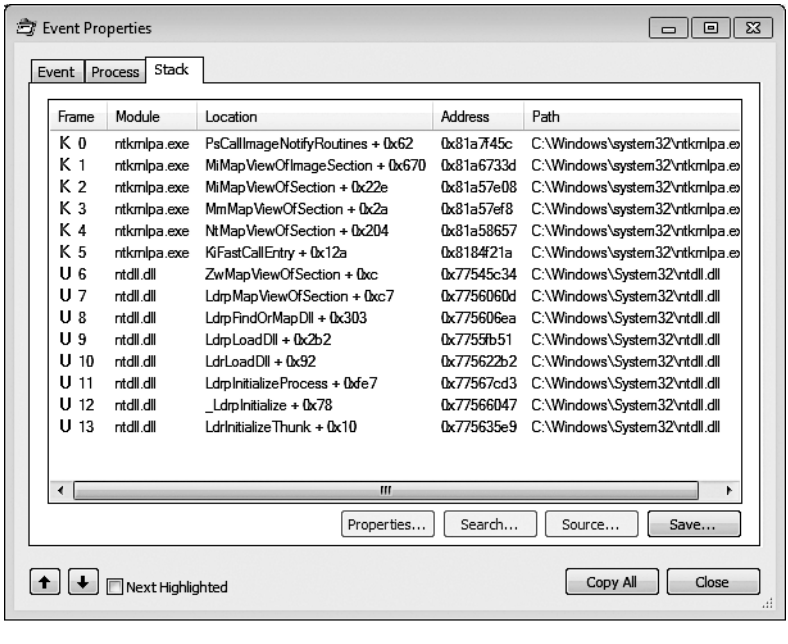
На следующей копии экрана показана дальнейшая череда событий, происходящих внутри самого процесса Notepad. Они инициированы в режиме ядра оболочкой запуска потока в режиме пользователя, которая выполняет ранее рассмотренные действия. Первые два представляют собой уведомительные сообщения отладки загрузчика образов Notepad.exe и Ntdll.dll, которые могут быть сгенерированы только сейчас, когда код запущен внутри контекста процесса Notepad, а не внутри контекста командной строки.



Затем свою лепту вносит механизм предвыборки, который ищет файл своей базы данных, который уже был создан процессом Notepad. На системе, где программа Блокнот уже запускалась хотя бы один раз, эта база данных будет в наличии, и механизм предвыборки начнет выполнение указанных в ней команд. Если это так, опустившись ниже, вы увидите несколько прочитанных и запрошенных DLL-библиотек. В отличие от обычной загрузки DLL-библиотеки, которая осуществляется загрузчиком образа в пользовательском режиме после просмотра таблиц импорта или когда приложение самостоятельно загружает DLL-библиотеку, эти события сгенерированы механизмом предвыборки, который уже знает о тех библиотеках, которые потребуются приложению Блокнот. Обычно за этим следует загрузка образа требуемых DLL-библиотек, и вы увидите события, подобные показанным на следующей копии экрана.



Теперь эти события сгенерированы из кода, запущенного внутри пользовательского режима, который был вызван, как только была завершена работа функции оболочки режима ядра. Поэтому это первые события, исходящие из процедуры LdrpInitializeProcess, которая, как мы уже упоминали, является внутренней системной функцией оболочки для любого нового процесса, пока не будет вызвана оболочка стартового адреса. Вы можете сами в этом убедиться, взглянув на стек этих событий, например на стек события загрузки образа kernel32.dll, показанный на следующей копии экрана.



Последующие события генерируются этой процедурой и ее вспомогательными функциями до тех пор, пока вы в конечном итоге не придете к событиям, сгенерированным функцией WinMain внутри Notepad, где код уже будет выполняться под управлением, заданным разработчиком. Подробное описание всех событий и компонентов пользовательского режима, которые вступают в игру в ходе выполнения процесса, заняли бы всю эту главу, поэтому исследование любых последующих событий предлагается в качестве упражнения для читателя этой книги. ■

Внутреннее устройство потоков

После того как мы препарировали процессы, обратим свое внимание на структуру потоков. Пока в явном виде не будет утверждаться обратное, вы можете считать, что все в этом разделе применимо как к потокам пользовательского режима, так и к системным потокам ядра (которые рассматривались в главе 2).

Структура данных

На уровне операционной системы поток Windows представлен создаваемым в исполняющей системе объектом потока. Этот объект инкапсулирует структуру ETHREAD, которая, в свою очередь, содержит в качестве своей первой составляющей структуру KTHREAD. Это показано на рис. 5.8. Структура ETHREAD и другие структуры, на которые она указывает, находятся в системном адресном пространстве, за исключением блока переменных окружения потока — thread environment block (TEB), — который находится в адресном пространстве процесса (опять-таки потому, что в доступе к нему нуждаются компоненты, выполняющиеся в пользовательском режиме).

Процесс подсистемы Windows (Csrss) поддерживает параллельную структуру для каждого потока, созданного в приложении этой подсистемы, которая называется CSR_THREAD. Для потоков, вызвавших USER- или GDI-функции подсистемы Windows, та часть этой подсистемы, которая выполняется в режиме ядра (Win32k.sys), поддерживает структуру данных для каждого потока (которая называется W32THREAD), на которую указывает структура KTHREAD.

ПРИМЕЧАНИЕ

Тот факт, что на исполняющую, высокоуровневую, связанную с графикой структуру потока Win32k указывает KTHREAD, а не ETHREAD, является нарушением системы уровней или упущением в стандартной архитектуре абстракции ядра — планировщик и другие низкоуровневые компоненты этим полем не пользуются.

Большинство полей, показанных на рис. 5.8, говорят сами за себя. Первой составляющей ETHREAD, которая называется Tcb (Thread control block — Блок управления потоком), является структура типа KTHREAD. Далее следует информация, касающаяся идентификации потока, информация, касающаяся идентификации процесса (включая указатель на процесс-владелец, чтобы можно было получить доступ к его информации окружения), информация безопасности в форме указателя на маркер доступа и информации о заимствовании прав,

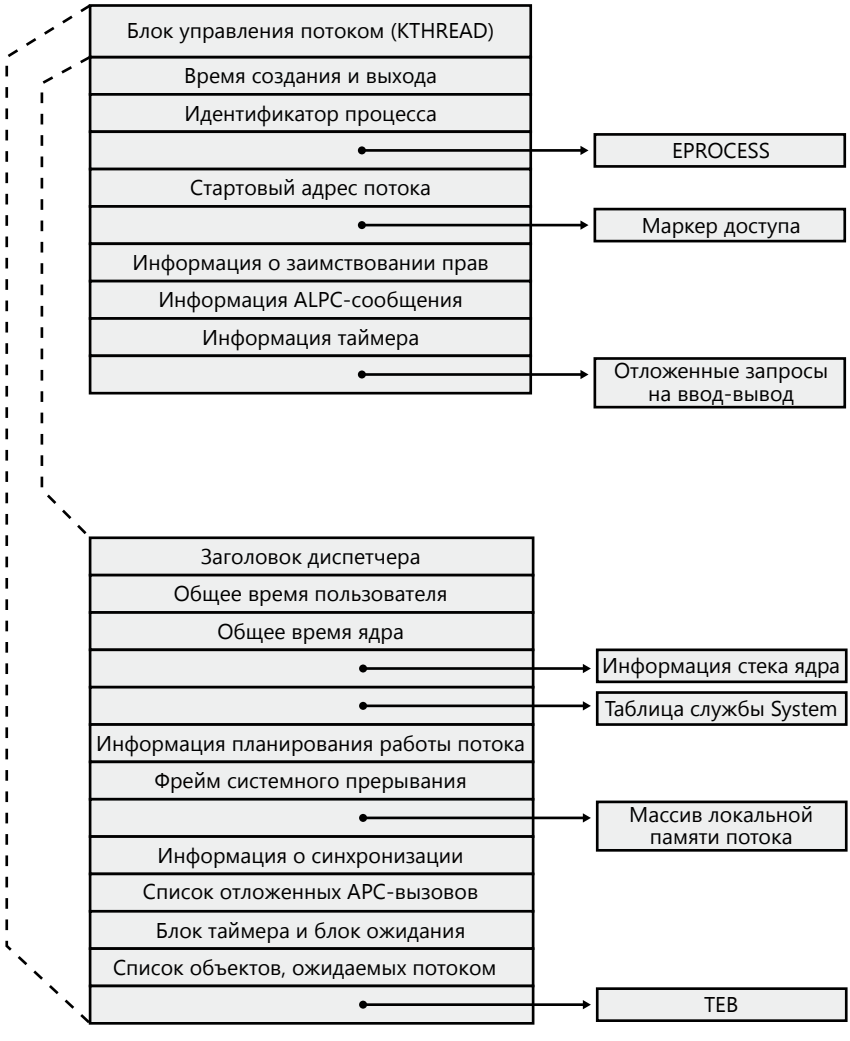


Рис. 5.8. Основные поля структуры потока в исполняющей системе и его встроенной структуры потока в ядре

и наконец, поля, относящиеся к сообщениям асинхронного вызова локальной процедуры — Asynchronous Local Procedure Call (ALPC) и к отложенным запросам на ввод-вывод. Некоторые из этих основных полей более подробно рассматриваются в других местах данной книги. Для более подробного изучения внутреннего устройства структуры ETHREAD можно воспользоваться командой отладчика ядра dt, которая отобразит ее формат.

Рассмотрим подробнее две основные структуры данных, упомянутые в предыдущем тексте: KTHREAD и TEB. Структура KTHREAD (которая является Tcb-составляющей структуры ETHREAD) содержит информацию, нужную ядру Windows для планирования выполнения потока, синхронизации и хронометража.

ЭКСПЕРИМЕНТ: ВЫВОД СТРУКТУР ETHREAD И KTHREAD

Структуры ETHREAD и KTHREAD можно вывести на экран с помощью команды dt отладчика ядра. В следующем выводе показан формат структуры ETHREAD на 32-разрядной системе:

```
lkd> dt nt!_ethread
nt!_ETHREAD
   +0x000 Tcb              : _KTHREAD
   +0x1e0 CreateTime       : _LARGE_INTEGER
   +0x1e8 ExitTime         : _LARGE_INTEGER
   +0x1e8 KeyedWaitChain   : _LIST_ENTRY
   +0x1f0 ExitStatus       : Int4B
...
   +0x270 AlpcMessageId           : Uint4B
   +0x274 AlpcMessage             : Ptr32 Void
   +0x274 AlpcReceiveAttributeSet : Uint4B
   +0x278 AlpcWaitListEntry       : _LIST_ENTRY
   +0x280 CacheManagerCount       : Uint4B
```

Структуру KTHREAD можно вывести с помощью аналогичной команды или путем набора команды dt nt!_ETHREAD Tcb, как было показано ранее в эксперименте со структурами EPROCESS и KPROCESS:

```
lkd> dt nt!_kthread
nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 CycleTime        : Uint8B
   +0x018 HighCycleTime    : Uint4B
   +0x020 QuantumTarget    : Uint8B
...
   +0x05e WaitIrql         : UChar
   +0x05f WaitMode         : Char
   +0x060 WaitStatus       : Int4B
```

■

ЭКСПЕРИМЕНТ: ИСПОЛЬЗОВАНИЕ КОМАНДЫ ОТЛАДЧИКА ЯДРА !THREAD

Команда отладчика ядра !thread выводит дамп поднабора информации в структурах данных потока. Некоторые основные элементы информации, выводимые отладчиком ядра, не могут быть выведены какой-либо утилитой, в том числе это внутренние адреса структуры, подробные данные о приоритете, информация о стеке, список отложенных запросов на ввод-вывод и для потоков, находящихся в состоянии ожидания, список объектов, ожидаемых потоком.

Для вывода информации о потоке нужно воспользоваться либо командой !process (которая выводит все потоки процесса после вывода информации о процессе), либо командой !thread с адресом объекта потока для вывода информации о конкретном потоке. ■

Структура TEB, показанная на рис. 5.9, является одной из структур, рассмотренных в данном разделе, которая находится в адресном пространстве процесса

(а не в системном адресном пространстве). Внутренне она состоит из заголовка, который называется блоком информации потока — TIB (Thread Information Block), чье существование обусловлено главным образом обеспечением совместимости с приложениями OS/2 и Win9x. Он также дает возможность хранить в менее объемной структуре информацию об исключениях и стеке при создании новых потоков с использованием исходного TIB (Initial TIB).

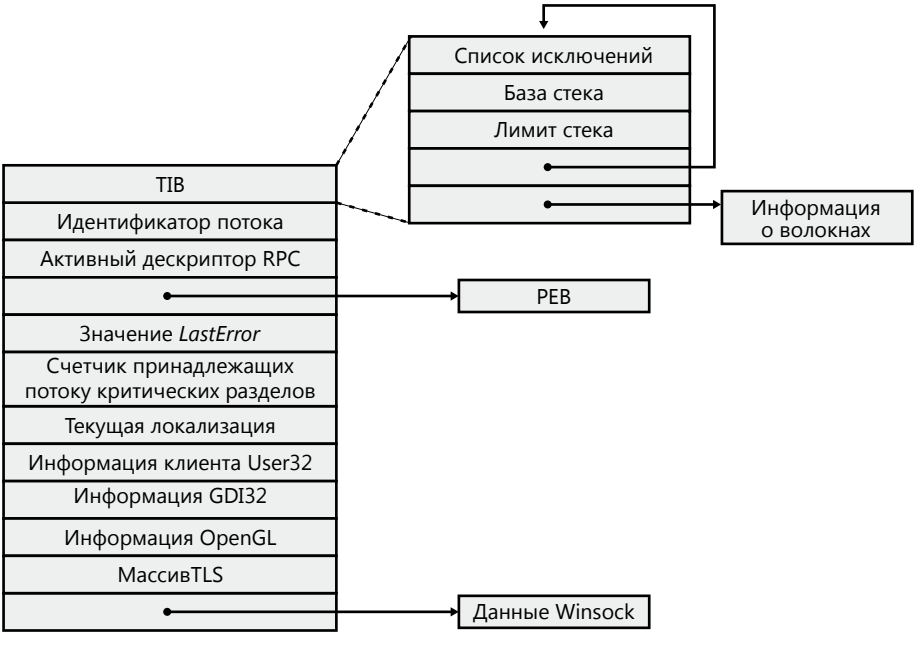


Рис. 5.9. Поля блока переменных окружения потока

ЭКСПЕРИМЕНТ: ПРОСМОТР ИНФОРМАЦИИ О ПОТОКЕ

Следующий вывод является подробным отображением процесса, созданным за счет использования утилиты Tlist из отладочного инструментария Debugging Tools for Windows. Следует заметить, что в списке потоков есть элемент Win32StartAddr. Это адрес, переданный функции CreateThread приложением. Все остальные утилиты, за исключением Process Explorer, которые показывают стартовый адрес потока, дают фактический стартовый адрес (функции в Ntdll.dll), а не стартовый адрес, указанный приложением.

```
C:\Program Files\Windows Kits\8.0\Debuggers\x86>tlist winword
3232 WINWORD.EXE       648739_Chap05.docx - Microsoft Word
   CWD:     C:\Users\Alex Ionescu\Documents\
   CmdLine: "C:\Program Files\Microsoft Office\Office14\WINWORD.EXE" /n "C:\Users\Alex
Ionescu\Documents\Chapter5.docx
   VirtualSize:   531024 KB   PeakVirtualSize:   585248 KB
   WorkingSetSize:122484 KB   PeakWorkingSetSize:181532 KB
   NumberOfThreads: 12
```

```
  2104 Win32StartAddr:0x2fde10ec LastErr:0x00000000 State:Waiting
  2992 Win32StartAddr:0x7778fd0d LastErr:0x00000000 State:Waiting
  3556 Win32StartAddr:0x3877e970 LastErr:0x00000000 State:Waiting
  2436 Win32StartAddr:0x3877e875 LastErr:0x00000000 State:Waiting
  3136 Win32StartAddr:0x3877e875 LastErr:0x00000000 State:Waiting
  3412 Win32StartAddr:0x3877e875 LastErr:0x00000000 State:Waiting
  1096 Win32StartAddr:0x3877e875 LastErr:0x00000000 State:Waiting
   912 Win32StartAddr:0x74497832 LastErr:0x00000000 State:Waiting
  1044 Win32StartAddr:0x389b0926 LastErr:0x00000583 State:Waiting
  1972 Win32StartAddr:0x694532fb LastErr:0x00000000 State:Waiting
  4056 Win32StartAddr:0x75f9c83e LastErr:0x00000000 State:Waiting
  1124 Win32StartAddr:0x777903e9 LastErr:0x00000000 State:Waiting
14.0.5123.5000 shp  0x2FDE0000  C:\Program Files\Microsoft Office\Office14\WINWORD.EXE
6.1.7601.17725 shp  0x77760000  C:\Windows\SYSTEM32\ntdll.dll
6.1.7601.17651 shp  0x75CE0000  C:\Windows\system32\kernel32.dll
```

■

В TEB хранится контекстная информация для загрузчика образов и различных DLL-библиотек Windows. Поскольку эти компоненты запускаются в пользовательском режиме, им нужна структура данных, в которую можно вести запись в этом режиме. Поэтому данная структура находится в адресном пространстве процесса, а не в системном пространстве, где в нее велась бы запись только из режима ядра. Адрес TEB можно определить с помощью команды отладчика ядра !thread.

ЭКСПЕРИМЕНТ: ИЗУЧЕНИЕ TEB

Дамп структуры TEB можно вывести с помощью команды отладчика ядра !teb. Он имеет следующий вид:

```
kd> !teb
TEB at 7ffde000
    ExceptionList:        019e8e44
    StackBase:            019f0000
    StackLimit:           019db000
    SubSystemTib:         00000000
    FiberData:            00001e00
...
    PEB Address:          7ffd9000
    LastErrorValue:       0
    LastStatusValue:      c0000139
    Count Owned Locks:    0
    HardErrorMode:        0
```

■

Структура CSR_THREAD, показанная на рис. 5.10, аналогична структуре данных CSR_PROCESS, но применяется к потокам. Как вы, наверное, помните, она поддерживается каждым Csrss-процессом внутри сеанса и идентифицирует потоки подсистемы Windows, запущенные внутри этого процесса. В структуре CSR_THREAD хранится дескриптор, который Csrss содержит для потока, различные флаги и указатель для потока на структуру CSR_PROCESS. Там также хранится еще одна копия времени создания потока.

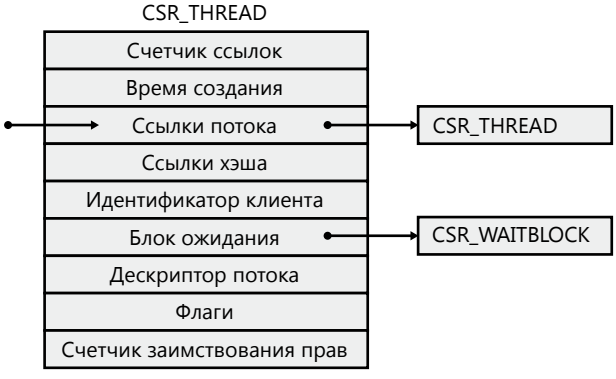


Рис. 5.10. Поля структуры CSR-потока

ЭКСПЕРИМЕНТ: ИЗУЧЕНИЕ CSR_THREAD

Дамп структуры CSR_THREAD можно вывести с помощью команды отладчика пользовательского режима !dt, подключившись к процессу Csrss. Чтобы выполнить эту операцию, следуйте инструкциям, которые были даны ранее в эксперименте со структурой CSR_PROCESS. Вывод имеет следующий вид:

```
0:000> !dt v 001c7630
PCSR_THREA @ 001c7630:
   +0x000 CreateTime       : _LARGE_INTEGER 0x1cb9fb6'00f90498
   +0x008 Link             : _LIST_ENTRY [ 0x1c0ab0 - 0x1c0f00 ]
   +0x010 HashLinks        : _LIST_ENTRY [ 0x75f19b38 - 0x75f19b38 ]
   +0x018 ClientId         : _CLIENT_ID
   +0x020 Process          : 0x001c0aa0 _CSR_PROCESS
   +0x024 ThreadHandle     : 0x000005c4
   +0x028 Flags            : 0
   +0x02c ReferenceCount   : 1
   +0x030 ImpersonateCount : 0
```

И наконец, структура W32THREAD, показанная на рис. 5.11, является аналогом структуры данных WIN32PROCESS, но применительно к процессам. Эта

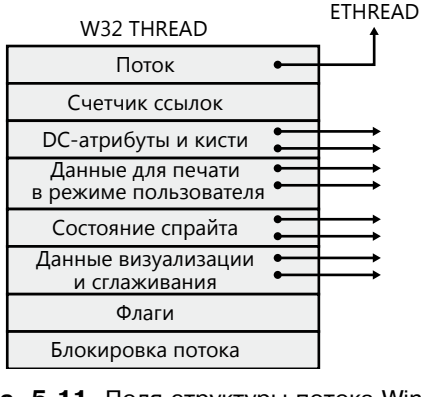


Рис. 5.11. Поля структуры потока Win32k

структура содержит главным образом информацию, полезную для подсистемы GDI (кисти и DC-атрибуты), а также для среды драйвера принтера пользовательского режима — User Mode Print Driver framework (UMPD), которая используется производителями для записи драйверов принтеров пользовательского режима. И наконец, она содержит состояние визуализации для компоновки рабочего стола.

ЭКСПЕРИМЕНТ: ИЗУЧЕНИЕ СТРУКТУРЫ W32THREAD

Вывести дамп структуры W32THREAD можно путем просмотра вывода команды !thread, в котором в поле Win32Thread дан указатель на эту структуру. Кроме того, если вы используете команду dt, в блоке KTHREAD есть поле под названием Win32Thread, в котором содержится указатель на эту структуру. Следует напомнить, что структура W32THREAD имеется только у GUI-потока, поэтому вполне возможно, что определенные потоки, например фоновые или рабочие, не будут иметь связанную с ними структуру W32THREAD. Поскольку каких-либо расширенных средств для просмотра структуры W32THREAD не существует, следует воспользоваться командой dt:

```
dt win32k!_w32thread ffb79dd8
   +0x000 pEThread         : 0x83ad4b60 _ETHREAD
   +0x004 RefCount         : 1
   +0x008 ptlW32           : (null)
   +0x00c pgdiDcattr       : 0x00130740
   +0x010 pgdiBrushAttr    : (null)
   +0x014 pUMPDObjs        : (null)
   +0x018 pUMPDHeap        : (null)
   +0x01c pUMPDObj         : (null)
...
   +0x0a8 bEnableEngUpdateDeviceSurface : 0 ''
   +0x0a9 bIncludeSprites  : 0 ''
   +0x0ac ulWindowSystemRendering : 0
```

■

Рождение потока

Жизненный цикл потока начинается тогда, когда программа создает новый поток. Запрос перекочевывает исполняющей системе Windows, где диспетчер процесса выделяет пространство под объект потока и вызывает ядро для инициализации блока управления потока (KTHREAD). Действия из следующего списка предпринимаются внутри Windows-функции CreateThread в библиотеке Kernel32.dll для создания потока Windows:

1. Функция CreateThread преобразует параметры Windows API к более подходящим флагам и создает соответствующую структуру, описывающую параметры объекта (OBJECT_ATTRIBUTES).
2. Функция CreateThread создает список атрибутов с двумя записями: идентификатором клиента — client ID и адресом TEB. Это позволяет функции CreateThread получить эти значения, как только поток будет создан.
3. Вызывается функция NtCreateThreadEx, предназначенная для создания контекста пользовательского режима, а также для исследования и сбора списка

атрибутов. Затем она вызывает функцию PspCreateThread для создания приостановленного объекта потока исполняющей системы (см. раздел «Порядок работы функции CreateProcess», описания этапов 3 и 5).

4. Функция CreateThread выделяет контекст активации для потока, используемый поддержкой смежной сборки (side-by-side assembly). Затем она запрашивает стек активации, чтобы посмотреть, не требует ли он активации, и проводит активацию в случае необходимости. Указатель на стек активации сохраняется в TEB нового потока.
5. Функция CreateThread уведомляет подсистему Windows о новом потоке, и подсистема выполняет ряд настроечной работы для нового потока.
6. Вызвавшему коду возвращается дескриптор и идентификатор потока, сгенерированные на этапе 3.
7. За исключением того случая, когда вызывающий код создал поток с установленным флагом создания приостановленного потока —CREATE_SUSPENDED, теперь поток возобновляется и может быть запланирован для выполнения. Когда поток начинает работать, он перед вызовом кода по фактическому стартовому адресу, указанному пользователем, выполняет действия, описанные ранее в разделе «Этап 7. Выполнение инициализации процесса в контексте нового процесса».

Изучение активности потока

Изучение активности потока играет очень важную роль, если вы пытаетесь определить, почему запущен или почему завис процесс, в котором выполняются сразу несколько служб (например, Svchost.exe, Dllhost.exe или Lsass.exe).

Различные элементы состояния Windows-потоков могут показать несколько инструментальных средств: WinDbg (в подключенном процессе режима пользователя и в режиме отладки ядра), монитор производительности (Performance Monitor) и Process Explorer (см. раздел «Планирование потоков»).

Для просмотра потоков в процессе с помощью Process Explorer выберите процесс и откройте свойства процесса (дважды щелкнув на имени процесса или щелкнув на пунктах меню Process (Процесс), Properties (Свойства)). Затем щелкните на вкладке Threads (Потоки). В этой вкладке показывается список потоков в процессе и четыре столбца информации. Для каждого потока там показываются идентификатор (ID) потока, процент потребления ресурса центрального процессора (на основе настроенного интервала обновлений), количество циклов, приходящихся на поток, и стартовый адрес потока. Информацию можно отсортировать по любому из этих четырех столбцов.

Вновь созданные потоки выделены зеленым цветом, а существовавшие потоки выделены красным. Продолжительность выделения может быть настроена с использованием пунктов меню Options (Настройки), Difference Highlight Duration (Изменение продолжительности выделения). Это может пригодиться для выявления создания в процессе ненужных потоков (потоки должны создаваться при запуске процесса, а не каждый раз, когда запрос обрабатывается внутри процесса).

При выборе в списке каждого потока Process Explorer показывает его идентификатор, время запуска, состояние, счетчик времени центрального процессора,

количество циклов, приходящихся на поток, количество переключений контекста, идеальный процессор и его группу, а также базовый и текущий приоритет. Есть также кнопка Kill (Уничтожить), с помощью которой можно завершить отдельный поток, но ею нужно пользоваться с большой осмотрительностью. Можно также воспользоваться кнопкой Suspend (Приостановить), с помощью которой сдерживается дальнейшее выполнение потока, предотвратив расход времени центрального процессора со стороны потока, вышедшего из-под контроля. Но это может привести к взаимным блокировкам, поэтому данной кнопкой нужно пользоваться с той же осмотрительностью, что и кнопкой Kill. И наконец, кнопка Permissions (Полномочия) позволяет просмотреть дескриптор безопасности потока (см. главу 6).

В отличие от Диспетчера задач и всех остальных средств отслеживания процессов и процессоров, Process Explorer использует счетчик тактовых импульсов, предназначенный для учета времени выполнения потока, а не таймер интервалов. Поэтому при использовании Process Explorer картина потребления ресурса центрального процессора будет существенно отличаться. Причина в том, что многие потоки запускаются на такой короткий отрезок времени, что они редко являются (если вообще являются) текущим запущенным потоком, когда происходит прерывание от интервального таймера. Поэтому они не нагружены основную часть их времени центрального процессора, это приводит к тому, что средства, основанные на использовании таймера, воспринимают использование центрального процессора равным 0 %. С другой стороны, общее количество тактовых циклов представляет фактическое количество циклов процессора, накопленное каждым потоком процесса. Оно не зависит от разрешения интервального таймера, поскольку счетчик поддерживается внутри системы процессором при каждом цикле и обновляется Windows при каждой записи прерывания. (Финальное накопление выполняется перед переключением контекста.)

Стартовый адрес потока выводится в форме «*модуль*!*функция*», где *модуль* — это имя файла с расширением .exe или .dll. Имя функции основывается на доступе к файлам символов модуля (см. главу 1, врезка «Эксперимент: просмотр подробностей процесса с помощью Process Explorer», с. 27). Если вы не уверены в происхождении модуля, щелкните на кнопке Module (Модуль). В Explorer откроется окно свойств файла того модуля, который содержит стартовый адрес потока (например, .exe или .dll).

ПРИМЕЧАНИЕ

Для потоков, созданных Windows-функцией CreateThread, Process Explorer показывает функцию, переданную CreateThread, а не фактическую стартовую функцию потока. Причина в том, что все потоки Windows стартуют в общей функции-оболочке запуска потока (RtlUserThreadStart в Ntdll.dll). Если Process Explorer показывал бы фактический стартовый адрес, оказалось бы, что большинство потоков процесса стартовало с одного и того же адреса, что вряд ли помогло бы разобраться, какой код выполнялся в потоке. Но если Process Explorer не может запросить адрес запуска, определенный пользователем (как в случает с защищенным процессом), он покажет функцию-оболочку, и вы увидите, что все потоки стартуют в функции RtlUserThreadStart.

Хотя в выведенном стартовом адресе потока может быть недостаточно информации для точного указания на то, чем занят поток и какой компонент внутри процесса отвечает за ресурс центрального процессора, потребляемый потоком. Это особенно характерно для случая, когда стартовым адресом потока является общая функция запуска (например, если имя функции ничего не говорит о том, чем этот поток занимается). Тогда ответ на этот вопрос можно получить, исследуя стек потока. Для просмотра стека потока дважды щелкните на интересующем вас потоке (или выделите этот поток и щелкните на кнопке Stack (Стек)). Process Explorer покажет стек потока (как пользовательского режима, так и режима ядра, если поток был запущен в режиме ядра).

ПРИМЕЧАНИЕ

В то время как отладчики пользовательского режима (WinDbg, Ntsd и Cdb) позволяют подключиться к процессу и вывести для потока стек пользовательского режима, Process Explorer показывает как пользовательский стек, так и стек ядра после простого щелчка на кнопке. Изучить стеки пользовательского потока и потока режима ядра можно также с помощью WinDbg в режиме локальной отладки ядра.

Просмотр стека потока может также помочь определить, почему процесс завис. В качестве примера на одной из систем через минуту после запуска зависло приложение Microsoft Office PowerPoint. Чтобы определить причину зависания, после запуска PowerPoint было использовано средство Process Explorer, чтобы изучить стек одного из потоков процесса. Результат показан на рис. 5.12.

Рис. 5.12. Стек зависшего потока в PowerPoint

Этот стек потока показывает, что PowerPoint (строка 10) вызвал функцию в Mso.dll, которая вызвала функцию OpenPrinterW в Winspool.drv. Затем Winspool.drv передал управление функции OpenPrinterRPC, которая вызвала функцию в DLL-библиотеке времени выполнения RPC, показывая, что был отправлен запрос удаленному принтеру. Итак, разбираться во внутреннем устройстве PowerPoint не понадобилось, имена модулей и функций, выведенные в стеке потока, показали, что поток ждал подключения к сетевому принтеру. На данной конкретной системе это был сетевой принтер, который не отвечал, чем и объяснялась задержка в запуске PowerPoint (приложения Microsoft Office подключаются ко всем сконфигурированным принтерам в процессе запуска). Подключение к этому принтеру было удалено из системы пользователя, и проблема была решена.

И наконец, при просмотре 32-разрядных приложений, запущенных на 64-разрядных системах в качестве процесса Wow64 (см. главу 3), Process Explorer по-

казывает для потоков как 32-разрядный, так и 64-разрядный стек. Поскольку ко времени системного вызова поток был переключен на 64-разрядный стек и контекст, простой просмотр 64-разрядного стека потока откроет только половину истории — 64-разрядную часть потока с кодом переключения Wow64. Поэтому при изучении процессов Wow64 нужно обязательно брать в расчет как 32-разрядный, так и 64-разрядный стек. Пример потока Wow64 внутри Microsoft Office Word 2007 показан на рис. 5.13. Выделенный фрейм стека и все фреймы стека, находящиеся ниже его, являются 32-разрядными фреймами стека из 32-разрядного стека. Фреймы стека, которые находятся выше выделенного фрейма, относятся к 64-разрядному стеку.

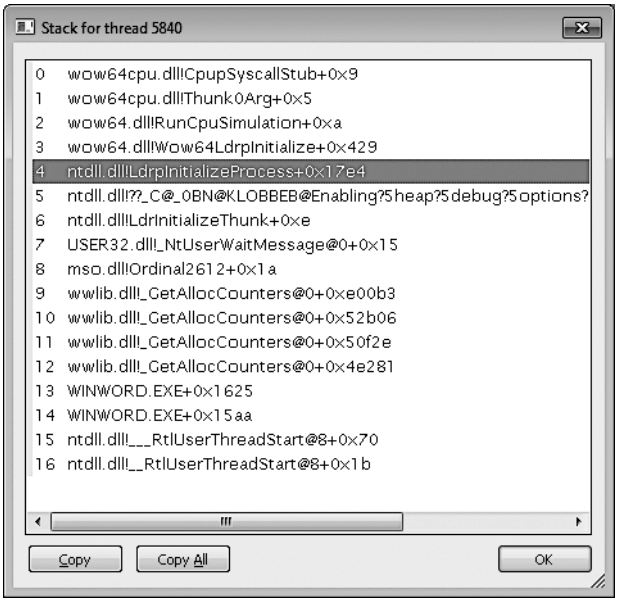


Рис. 5.13. Пример стека Wow64

Ограничения, накладываемые на потоки защищенного процесса

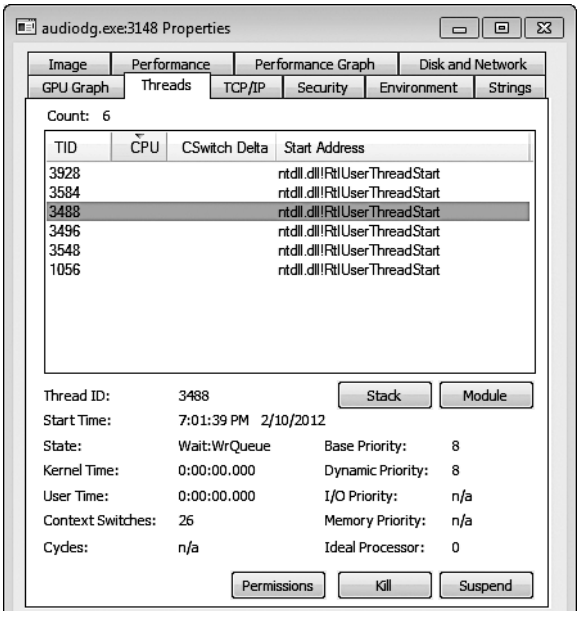
В разделе, посвященном внутреннему устройству процесса, уже упоминалось, что у защищенных процессов имеется ряд ограничений, под которые подпадали права доступа, которыми наделялись даже пользователи с самыми высокими привилегиями в системе. Эти ограничения также применяются к потокам внутри такого процесса. Тем самым гарантируется, что код, запущенный внутри защищенного процесса, не может быть взломан или каким-то иным образом затронут посредством стандартных Windows-функций, которые запрашивают права доступа, не предоставляемые потокам защищенных процессов. Фактически единственными предоставляемыми правами являются право на приостановку и возобновление выполнения потока — THREAD_SUSPEND_RESUME и право на получение по запросу установленной для потока ограниченной информации — THREAD_SET/QUERY_LIMITED_ INFORMATION.

ЭКСПЕРИМЕНТ: ПРОСМОТР ИНФОРМАЦИИ О ПОТОКЕ ЗАЩИЩЕННОГО ПРОЦЕССА

В предыдущем разделе было показано, как Process Explorer может использоваться при исследовании активности потока для определения причины потенциальных проблем системы или приложения. Теперь мы воспользуемся Process Explorer для того, чтобы посмотреть на защищенный процесс и увидеть, как различные права доступа, в которых было отказано, влияют на способности этого средства и на пользу от его применения в отношении таких процессов.

Найдите в списке процессов службу Audiodg.exe. Это процесс, отвечающий за основную часть внутренней работы, лежащей в основе аудиостека Windows, работающего в пользовательском режиме, и он нуждается в защите, чтобы гарантировать отсутствие утечки декодированного аудиоконтента высокого разрешения в ненадежные источники. Вызовите просмотр свойств процесса и посмотрите на вкладку Performance (Производительность). Посмотрите на количественные данные WS Private, WS Shareable и WS Shared, равные 0, при том что обо всем рабочем наборе данные по-прежнему отображаются. Это пример применения права THREAD_QUERY_INFORMATION в сравнении с правом THREAD_QUERY_LIMITED_INFORMATION.

Более того, загляните во вкладку Threads (Потоки). Там можно увидеть, что Process Explorer не может показать стартовый адрес потока Win32 и вместо этого показывает стартовый адрес стандартной оболочки потока, находящейся внутри Ntdll.dll. При попытке щелкнуть на кнопке Stack (Стек) вы получите сообщение об ошибке, потому что Process Explorer нужно прочитать виртуальную память внутри защищенного процесса, чего он не может сделать.



И наконец, обратите внимание на то, что наряду с отображением приоритетов базовых и динамических приоритетов — Base и Dynamic, приоритеты ввода вывода и памяти — I/O и Memory не показаны, что является

еще одним примером ограничений по сравнению с полной информацией, получаемой по запросу прав доступа. При попытке уничтожения потока внутри Audiodg.exe вы заметите еще одно сообщение об ошибке отказа в доступе, напоминающее об отсутствии права на завершение потока — THREAD_TERMINATE. ■

Рабочие фабрики (пулы потоков)

Термин «рабочие фабрики» относится к внутреннему механизму, используемому для реализации пулов потоков пользовательского режима. Устаревшие процедуры пула потоков были полностью реализованы в пользовательском режиме внутри библиотеки Ntdll.dll, и Windows API предоставляет различные процедуры для вызова соответствующих подпрограмм, обеспечивающих таймеры ожидания, ожидающие функции обратного вызова и автоматическое создание и удаление потоков в зависимости от объема проделываемой работы.

Поскольку ядро может иметь непосредственный контроль над планированием, созданием и завершением потоков без обычных издержек, связанных с выполнением этих операций из пользовательского режима, большинство функциональных механизмов, требующихся для поддержки реализации в Windows пула потоков пользовательского режима, теперь располагаются в ядре, что также упрощает код, который нужно создавать разработчикам. Например, создание рабочего пула в удаленном процессе может быть осуществлено с помощью одного-единственного API-вызова вместо сложной череды вызовов в виртуальной памяти, которые обычно для этого требовались. Согласно этой модели, библиотека Ntdll.dll просто предоставляет интерфейсы и высокоуровневые API-функции, требуемые для создания интерфейса с кодом рабочей фабрики.

Этот механизм пула потоков, управляемого ядром, управляется с помощью такого типа диспетчера объектов, который называется TpWorkerFactory, а также с помощью четырех исходных системных вызовов для управления фабрикой и ее работниками (NtCreateWorkerFactory, NtWorkerFactoryWorkerReady, NtReleaseWorkerFactoryWorker, NtShutdownWorkerFactory), двух исходных вызовов запросов-установок (NtQueryInformationWorkerFactory и NtSetInformationWorkerFactory) и ждущего вызова (NtWaitForWorkViaWorkerFactory).

Как и все другие исходные системные вызовы, эти вызовы предоставляют пользовательскому режиму дескриптор объекта TpWorkerFactory, в котором содержится такая информация, как имя и атрибуты объекта, заданная маска доступа и дескриптор безопасности. Но в отличие от других системных вызовов, упакованных Windows API, управление пулом потоков обрабатывается исходным кодом библиотеки Ntdll.dll, это означает, что разработчики работают с непроницаемым дескриптором (указателем TP_WORK), принадлежащем Ntdll.dll, в котором хранится настоящий дескриптор.

Как следует из названия «рабочая фабрика», реализация этой фабрики отвечает за размещение рабочих потоков (и за вызов кода в заданной точке входа рабочего потока пользовательского режима), поддержку минимального и максимального числа потоков (для постоянных рабочих пулов, либо для полностью динамических пулов), а также за другую учетную информацию. Это позволяет выполнять такие операции, как завершение работы пула потоков с помощью единственного обращения к ядру, поскольку ядро является единственным компонентом, отвечающим за создание и завершение потоков.

Поскольку ядро создает новые потоки по мере надобности в динамическом режиме, основываясь на предоставленных минимальных и максимальных количествах, это также повышает масштабируемость приложений, использующих новую реализацию пула потоков. Рабочая фабрика создаст новый поток при выполнении всех следующих условий:

- Количество доступных работником ниже максимального количества работников, указанного для фабрики (по умолчанию это количество составляет 500 работников).
- Рабочая фабрика имеет привязанные к ней объекты (привязанным объектом может быть, к примеру, ALPC-порт, на котором строит свое ожидание рабочий поток) или поток, активированный в пуле.
- Существуют отложенные пакеты запросов ввода-вывода (IRP-пакеты), связанные с рабочим потоком.
- Разрешено создание динамических потоков.

И она завершит работу потоков при простое свыше 10 секунд (по умолчанию).

Кроме того, при старой реализации разработчики всегда могли воспользоваться максимально возможным количеством потоков, основанном на количестве процессоров в системе. Благодаря поддержке динамических процессоров в Windows Server теперь есть возможность использовать для приложений пулы потоков для автоматического использования новых процессоров, добавленных в ходе выполнения кода.

Следует заметить, что поддержка рабочей фабрики является всего лишь оболочкой для управления обычными задачами, которая в противном случае должна была бы выполняться в пользовательском режиме (с потерей производительности). Основная часть логики нового кода пула потоков остается в той стороне этой архитектуры, которая относится к библиотеке Ntdll.dll. (Теоретически, используя недокументированные функции для рабочих фабрик, можно создать другую реализацию пула потоков.) К тому же это не код рабочих фабрик предоставляет масштабируемость, внутренние механизмы ожидания и эффективность обработки рабочих заданий. Этим занимается куда более старый компонент Windows, который уже рассматривался — порт завершения ввода-вывода, или, если быть точнее, очередей ядра — kernel queues (KQUEUE).

Фактически, при создании рабочей фабрики должен быть уже создан код пользовательского режима и должен быть передан его дескриптор. Именно через этот порт завершения ввода-вывода реализация пользовательского режима будет выстраивать в очередь на работу, а также находиться в ожидании работы путем использования системных вызовов рабочей фабрики вместо API-функций порта завершения ввода-вывода. Но, согласно внутреннему устройству, «освобождающий» вызов рабочей фабрики (который ставит работу в очередь) является оболочкой вокруг процедуры IoSetIoCompletionEx, которая увеличивает количество отложенной работы, а «ожидающий» вызов является оболочкой вокруг процедуры IoRemoveIoCompletion. Обе эти процедуры совершают вызов в код реализации очереди ядра.

Задача кода рабочей фабрики заключается в управлении либо постоянными, статическими, либо динамическими пулами потоков, в создании оболочки модели порта завершения ввода-вывода в интерфейсах, пытающихся предотвратить

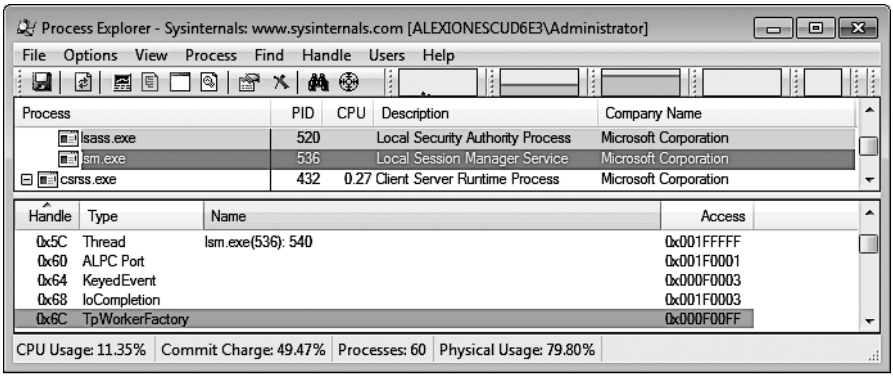
остановку продвижения рабочих очередей путем автоматического создания динамических потоков, и в упрощении глобальной очистки и завершении операций в ходе запроса на завершение работы фабрики, а также в беспрепятственной блокировке новых запросов к фабрике при таком развитии событий.

К сожалению, структура данных, используемая реализацией рабочей фабрики, не выражается в виде общедоступных символов, но, как показано в следующем эксперименте, возможность посмотреть на некоторые рабочие пулы все же есть. Кроме того, API-функция NtQueryInformationWorkerFactory выводит дамп почти каждого поля в структуре рабочей фабрики.

ЭКСПЕРИМЕНТ: ПОСМОТР ПУЛОВ ПОТОКОВ

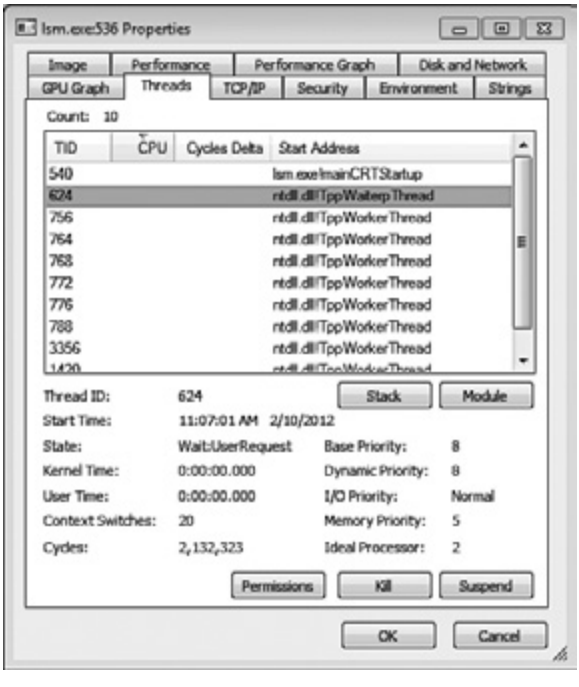
Из-за преимуществ использования механизма пула потоков ими пользуются многие основные системные компоненты и приложения, особенно когда они работают с такими ресурсами, как ALPC-порты (для динамической обработки входящих сообщений на соответствующем и масштабируемом уровне). Одним из способов определения тех процессов, которые используют рабочую фабрику, является просмотр списка дескрипторов в Process Explorer. Чтобы посмотреть на некоторые присущие им особенности, выполните следующие действия:

1. Запустите Process Explorer и в меню View (Вид) установите флажок Show Unnamed Handles And Mappings (Показывать безымянные дескрипторы и отображения). К сожалению, библиотека Ntdll.dll не дает имена рабочим фабрикам, что и определяет необходимость данного действия с целью просмотра дескрипторов.
2. В списке процессов выберите Lsm.exe и посмотрите на таблицу дескрипторов. Убедитесь в том, что включено отображение нижней панели (View (Вид), Show Lower Pane (Показывать нижнюю панель)) и она находится в режиме показа таблицы дескрипторов (View (Вид), Lower Pane View (Вид нижней панели), Handles (Дескрипторы)).
3. Щелкните правой кнопкой мыши на столбцах панели, а затем выберите пункт Select Columns (Выбрать столбцы). Убедитесь в том, что для отображения выбран столбец Type (Тип), и щелкните на нем, чтобы отсортировать список по типам.
4. Теперь опускайтесь вниз по списку дескрипторов, наблюдая за содержимым столбца Type (Тип), пока не найдете дескриптор типа TpWorkerFactory.



Обратите внимание на то, что дескриптору TpWorkerFactory непосредственно предшествует дескриптор IoCompletion (в порядковом отношении; чтобы это увидеть, отсортируйте список по столбцу «Handle» (Дескриптор)). Как уже ранее было замечено, причиной этому служит то, что перед созданием рабочей фабрики должен быть создан дескриптор порта завершения ввода-вывода, на который будет послана работа.

5. Теперь в списке процессов дважды щелкните на процессе Lsm.exe и щелкните на вкладке Threads (Потоки).



На этой системе (с двумя процессорами) рабочая фабрика создала шесть рабочих потоков по запросу Lsm.exe (процессы могут определить минимальное и максимальное количество потоков) и основывалась при этом на использовании процесса и количества процессоров на машине. Эти потоки идентифицируются как TppWorkerThread, что является при осуществлении системных вызовов рабочей фабрики рабочей точкой входа библиотеки Ntdll.dll.

6. Библиотека Ntdll.dll отвечает за свой собственный внутренний учет в оболочке рабочего потока (TppWorkerThread) перед вызовом рабочей функции обратного вызова, зарегистрированной приложением. Путем просмотра причины ожидания — Wait в информации о состоянии — State для каждого потока, можно выстроить примерное предположение о том, чем занят каждый рабочий поток. Щелкните дважды на одном из потоков внутри LPC-ожидания, чтобы просмотреть его стек (см. рис. на с. 475).

Конкретно этот рабочий поток был использован Lsm.exe для LPC-связи. Поскольку диспетчеру локального сеанса требуется обмениваться данными посредством LPC с другими компонентами, такими как Smss и Csrss, вполне возможно, что ему может понадобиться загрузить несколько своих

потоков ответами на сообщения LPC и ожиданиями этих сообщений. (Чем больше занимающихся этим потоков, тем меньше потери в скорости работы конвейера LPC.)

Если посмотреть на другие рабочие потоки, можно увидеть, что некоторые из них ожидают появления таких объектов, как события. У процесса может быть несколько пулов потоков, и каждый пул потоков может иметь различные потоки, выполняющие совершенно не связанные друг с другом задачи. Ту или иную работу назначает разработчик, который вызывает API-функции пула потоков для регистрации этой работы посредством Ntdll.dll.

Планирование потоков

В этом разделе рассматриваются политика и алгоритмы Windows-планирования. В первом подразделе дается краткое описание порядка планирования работы в Windows и определения основным понятиям. Затем дается описание уровней приоритета как с точки зрения Windows API, так и с точки зрения ядра Windows. После обзора соответствующих утилит и инструментальных средств Windows, имеющих отношение к планированию, будут подробно представлены структуры данных и алгоритмы, составляющие систему планирования Windows, включая описание общих сценариев планирования и порядка выбора потока, а также выбора процессора.

Обзор организации планирования в Windows

В Windows реализуется приоритетная, вытесняющая система планирования, при которой всегда выполняется хотя бы один работоспособный (готовый) поток

с самым высоким приоритетом, с той оговоркой, что конкретные, имеющие высокий приоритет и готовые к запуску потоки могут быть ограничены процессами, на которых им разрешено или предпочтительнее всего работать. Это явление называется *родственностью процессора*. Эта родственность определяется на основе заданной группы процессоров, в которую включается до 64 процессоров. По умолчанию потоки могут запускаться только на любом соответствующем процессоре из связанной с процессом группы (для обеспечения совместимости с устаревшими версиями Windows, поддерживающими только 64 процессора), но разработчики могут изменить родственность процессора, воспользовавшись соответствующими API-функциями или путем настройки маски родственности в заголовке образа, а пользователи могут воспользоваться инструментальными средствами для изменения родственности в ходе выполнения программы или при создании процесса. Но, хотя несколько потоков в процессе могут быть связаны с разными группами, сам по себе поток может запуститься только на процессорах, доступных внутри связанной с ним группы. Кроме того, разработчики могут выбрать создание знающих о группе приложений, использующих расширенные API-функции планирования, для связи логических процессоров из разных групп с родственностью их потоков. Это связывает процесс с несколькими группами, что теоретически позволит ему запускать свои потоки на любом доступном процессоре машины.

ЭКСПЕРИМЕНТ: ПРОСМОТР ГОТОВЫХ ПОТОКОВ

Список готовых потоков можно просмотреть с помощью команды отладчика ядра !ready. Эта команда выводит поток или список потоков, готовых к запуску на каждом уровне приоритетов. В следующем примере, сгенерированном на 32-разрядной машине с двухъядерным процессором, два потока готовы к запуску с приоритетом 8 на первом логическом процессоре и один поток с приоритетом 10, два потока с приоритетом 9 и три потока с приоритетом 8 готовы к запуску на втором логическом процессоре. Определение, которые из этих потоков будут запущены на своих соответствующих процессорах, сводится к простому выбору первого верхнего потока в очереди с самым высоким приоритетом (поток 857d9030 для логического процессора 0, и поток 857c0030 для логического процессора 1), но то, что очереди содержат именно эти потоки, является результатом выполнения нескольких довольно сложных алгоритмов, используемых планировщиком. Эта тема будет рассмотрена в данном разделе чуть позже.

```
kd> !ready
Processor 0: Ready Threads at priority 8
    THREAD 857d9030  Cid 0ec8.0e30  Teb: 7ffdd000 Win32Thread: 00000000 READY
    THREAD 855c8300  Cid 0ec8.0eb0  Teb: 7ff9c000 Win32Thread: 00000000 READY
Processor 1: Ready Threads at priority 10
    THREAD 857c0030  Cid 04c8.0378  Teb: 7ffdf000 Win32Thread: fef7f8c0 READY
Processor 1: Ready Threads at priority 9
    THREAD 87fc86f0  Cid 0ec8.04c0  Teb: 7ffd3000 Win32Thread: 00000000 READY
    THREAD 88696700  Cid 0ec8.0ce8  Teb: 7ffa0000 Win32Thread: 00000000 READY
Processor 1: Ready Threads at priority 8
    THREAD 856e5520  Cid 0ec8.0228  Teb: 7ff98000 Win32Thread: 00000000 READY
    THREAD 85609d78  Cid 0ec8.09b0  Teb: 7ffd9000 Win32Thread: 00000000 READY
    THREAD 85fdeb78  Cid 0ec8.0218  Teb: 7ff72000 Win32Thread: 00000000 READY
```

После того как поток был выбран для запуска, он запускается на время, называемое квантом. Квант — это продолжительность времени, в течение которого потоку разрешено работать, пока не настанет очередь запускаться другому потоку с тем же уровнем приоритета. Значение кванта может варьироваться от системы к системе и от процессора к процессору по любой из трех причин:

- Настроек конфигурации системы (длинные или короткие кванты, переменные или фиксированные кванты и разнос приоритетов).
- Состояния процесса, то есть является ли он процессом первого плана или фоновым процессом.
- Использования объекта задания для изменения кванта.

Более подробно все это будет рассмотрено далее в разделе «Кванты времени», а также в разделе «Объекты заданий».

Но поток может и не израсходовать свой квант времени, поскольку в Windows реализуется вытесняющий планировщик: если становится готов к запуску другой поток с более высоким приоритетом, текущий выполняемый поток может быть вытеснен еще до окончания его кванта времени. Фактически поток может быть выбран на запуск следующим и вытеснен еще даже до начала своего кванта времени!

Код планировщика Windows реализован в ядре. Но единого модуля или процедуры под названием «планировщик» не существует, код разбросан по ядру, где происходят события, связанные с планированием. Процедуры, выполняющие эти обязанности, обобщенно называются диспетчером ядра. Диспетчеризации потоков могут потребовать следующие события:

- Поток становится готовым к выполнению, например поток был только что создан или только что был освобожден от состояния ожидания.
- Поток выходит из состояния выполнения из-за окончания его кванта времени, его работа завершается, ему предоставляется возможность выполнения или он входит в состояние ожидания.
- Изменяется приоритет потока либо по причине вызова системной службы, либо по причине того, что Windows сама изменяет значение приоритета.
- Изменяется родственность процессора потока, и он больше уже не может быть запущен на том процессе, на котором выполнялся.

По совокупности всех этих обстоятельств Windows должна определить, какой из потоков должен быть запущен следующим на логическом процессоре, на котором работал поток, если это приемлемо, или на каком логическом процессоре не должен быть запущен поток. После того как логический процессор выбрал новый поток для выполнения, он в конечном итоге выполняет переключение контекста на этот поток. Переключение контекста представляет собой процедуру сохранения изменяющегося состояния процессора, связанного с запущенным потоком, загрузки изменяемого состояния другого потока и запуска выполнения нового потока.

Как уже отмечалось, Windows осуществляет планировку потоков по вытесняющему принципу. Такой подход имеет смысл, если учесть, что процессы не запускаются, а только предоставляют ресурсы и контекст, в котором запускаются их потоки. Поскольку решения по планированию принимаются исключительно

на основе потоков, не учитывается, какому процессу какой поток принадлежит. Например, если процесс А имеет 10 работоспособных потоков, процесс Б имеет 2 работоспособных потока и все 12 потоков имеют один и тот же приоритет, каждый поток теоретически получит одну двенадцатую времени центрального процессора, Windows не даст по 50 % процессорного времени процессу А и процессу Б.

Уровни приоритета

Чтобы разобраться в алгоритмах планирования потоков, сначала нужно понять, что такое уровни приоритета, используемые Windows. Как показано на рис. 5.14, внутренне Windows использует 32 уровня приоритета, от 0 до 31. Эти значения разбиваются на части следующим образом:

- шестнадцать уровней реального времени (от 16 до 31);
- шестнадцать изменяющихся уровней (от 0 до 15), из которых уровень 0 зарезервирован для потока обнуления страниц.

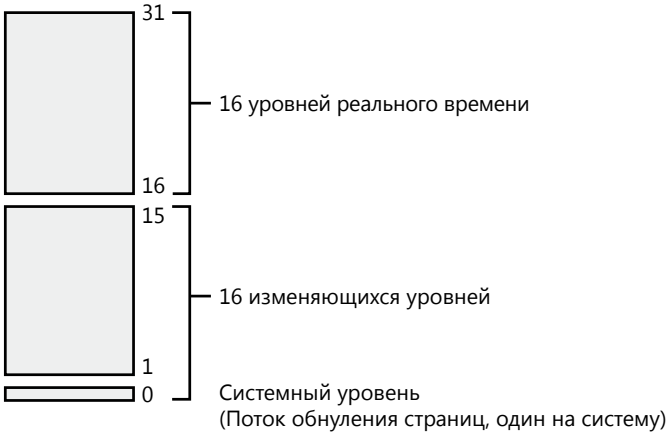


Рис. 5.14. Уровни приоритета потоков

Уровни приоритета потоков назначаются исходя из двух разных позиций: одной от Windows API и другой от ядра Windows. Сначала Windows API систематизирует процессы по классу приоритета, который им присваивается при создании (номера представляют внутренний индекс PROCESS_PRIORITY_CLASS, распознаваемый ядром): Реального времени — Real-time (4), Высокий — High (3), Выше обычного — Above Normal (7), Обычный — Normal (2), Ниже обычного — Below Normal (5) и Простоя — Idle (1).

Затем назначается относительный приоритет отдельных потоков внутри этих процессов. Здесь номера представляют изменение приоритета, применяющееся к базовому приоритету процесса: Критичный по времени — Time-critical (15), Наивысший — Highest (2), Выше обычного — Above-normal (1), Обычный — Normal (0), Ниже обычного — Below-normal (–1), Самый низший — Lowest (–2) и Простоя — Idle (–15).

Поэтому в Windows API каждый поток имеет базовый приоритет, являющийся функцией класса приоритета процесса и его относительного приоритета процесса. В ядре класс приоритета процесса преобразуется в базовый приоритет путем использования процедуры PspPriorityTable и показанных ранее индексов PROCESS_PRIORITY_CLASS, устанавливающих приоритеты 4, 8, 13, 14, 6 и 10 соответственно. (Это фиксированное отображение, которое не может быть изменено.) Затем применяется относительный приоритет потока в качестве разницы для этого базового приоритета. Например, наивысший «Highest»-поток получит базовый приоритет потока на два уровня выше, чем базовый приоритет его процесса.

Это отображение Windows-приоритета на внутренние номерные приоритеты Windows показано в табл. 5.3.

Таблица 5.3. Отображение приоритетов ядра Windows на Windows API

Класс приоритета/ Относительный приоритет	Realtime	High	Above	Normal	Below Normal	Idle
Time Critical (+ насыщение)	31	15	15	15	15	15
Highest (+2)	26	15	12	10	8	6
Above Normal (+1)	25	14	11	9	7	5
Normal (0)	24	13	10	8	6	4
Below Normal (–1)	23	12	9	7	5	3
Lowest (–2)	22	11	8	6	4	2
Idle (– насыщение)	16	1	1	1	1	1

Следует заметить, что относительные приоритеты потоков Time-Critical и Idle сохраняют свои соответствующие значения независимо от класса приоритета процесса (если только это не Realtime). Причина в том, что Windows API требует насыщения (saturation) приоритета от ядра путем передачи 16 или –16 в качестве запрошенного относительного приоритета (вместо 15 или –15). Тогда это будет распознано ядром как запрос на насыщение, и в структуре KTHREAD будет установлено поле Saturation. Это приведет к тому, что при положительном насыщении поток получит наивысший возможный приоритет внутри его класса приоритета (динамического или реального времени) или при отрицательном насыщении он получит самый низкий из возможных приоритетов. Кроме того, последующие запросы на изменение базового приоритета процесса уже не будут влиять на базовый приоритет этих потоков, потому что насыщенные потоки пропускаются в коде обработки.

В то время как у процесса имеется только одно базовое значение приоритета, у каждого потока имеется два значения приоритета: текущее и базовое. Решения по планированию принимаются исходя из текущего приоритета. Как поясняется в следующем разделе, посвященном повышениям приоритета, система при определенных обстоятельствах на короткие периоды времени повышает приоритет потоков в динамическом диапазоне (от 0 до 15). Windows никогда не регулирует приоритет потоков в диапазоне реального времени (от 16 до 31), поэтому они всегда имеют один и тот же базовый и текущий приоритет.

Исходный базовый приоритет потока наследуется из базового приоритета процесса. Процесс по умолчанию наследует свой базовый приоритет у того про-

цесса, который его создал. Это поведение может быть заменено другим в функции CreateProcess или путем использования пусковой команды в окне командной строки. Приоритет процесса может быть также изменен после создания процесса путем использования функции SetPriorityClass или различных инструментальных средств, предлагающих такую функцию, например Диспетчера задач и Process Explorer (щелчок правой кнопкой мыши на имени процесса и выбор нового класса приоритета). Например, можно снизить приоритет процесса, который интенсивно использует центральный процессор, чтобы он не мешал обычным действиям системы. Изменение приоритета процесса изменяют приоритеты потоков, повышая их или снижая, но их относительные установки остаются прежними.

Как правило, пользовательские приложения и службы запускаются с обычным базовым приоритетом (normal), поэтому их исходный поток чаще всего выполняется с уровнем приоритета 8. Но некоторые системные процессы Windows (например, диспетчер сеансов, диспетчер управления службами и процесс идентификации локальной безопасности) имеют немного более высокий базовый приоритет, чем тот, который используется по умолчанию для класса Normal (8). Это более высокое значение по умолчанию гарантирует, что все потоки в этих процессах будут запускаться с более высоким приоритетом, превышающим значение по умолчанию равное 8.

Приоритеты реального времени

Повысить или понизить приоритет потока в динамическом диапазоне можно в любом приложении, но у вас должны быть привилегии, позволяющие повышать приоритет, использующийся при планировании для ввода значения в пределах динамического диапазона. Следует иметь в виду, что многие важные системные потоки Windows, выполняемые в режиме ядра, работают в диапазоне приоритетов реального времени, поэтому если потоки тратят слишком много времени, работая в этом диапазоне, они могут заблокировать очень важные системные функции (например, диспетчер памяти, диспетчер кэша или какие-нибудь драйверы устройств).

Как только процесс входит в диапазон реального времени, все его потоки (даже потоки простоя) должны с помощью стандартных Windows API-функций запускаться на одном из уровней приоритета реального времени. Таким образом, становится невозможным смешивать потоки реального времени и динамические потоки. Причина в том, что API-функция SetThreadPriority вызывает присущую системе API-функцию NtSetInformationThread с информационным классом ThreadBasePriority, что позволяет приоритетам оставаться только в одном и том же диапазоне. Кроме того, этот информационный класс позволяет приоритету изменяться только в очевидном разбросе Windows API от –2 до 2 (или от реального времени до простоя), если только запрос не пришел от CSRSS или от процесса реального времени. Иными словами, это означает, что процесс реального времени имеет возможность подбирать приоритеты потоков где-то между 16 и 31, даже при том, что относительные приоритеты потоков Windows API вроде бы ограничивают его выбор на основе ранее показанной таблицы.

Но при вызове этих API-функций с информационным классом ThreadActualBasePriority для потока может быть непосредственно установлен базовый при-

оритет ядра, в том числе в динамическом диапазоне для процессов реального времени.

ПРИМЕЧАНИЕ

Как проиллюстрировано на рис. 5.15, где показываются уровни запроса прерываний — interrupt request levels (IRQL), хотя у Windows есть набор приоритетов, называемый приоритетами реального времени, в общем смысле этого понятия они не относятся к реальному времени. Причина в том, что Windows не предоставляет средства операционной системы реального времени, например гарантированные задержки прерывания или способ получения потоками гарантированного времени выполнения.

Сопоставление уровней прерывания с уровнями приоритета

На рис. 5.15 показаны уровни запроса прерываний (IRQL) для 32-разрядной системы. Потоки обычно запускаются на уровне IRQL 0 (который называется пассивным уровнем, потому что никакие прерывания не обрабатываются и никакие прерывания не заблокированы) или на уровне IRQL 1 (APC-уровень). Код пользовательского режима всегда запускается на пассивном уровне. Поэтому никакие потоки пользовательского уровня независимо от их приоритета не могут даже заблокировать аппаратные прерывания (хотя высокоприоритетные потоки реального времени могут заблокировать выполнение важных системных потоков).

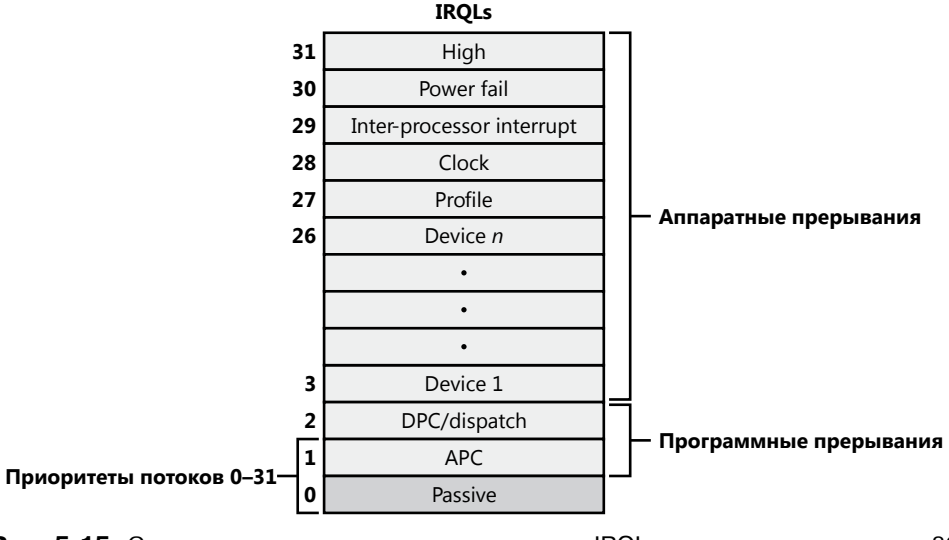


Рис. 5.15. Сопоставление приоритетов потоков с IRQL-уровнями на системе x86

Потоки, запущенные в режиме ядра, несмотря на изначальное планирование на пассивном уровне или уровне APC, могут поднять IRQL на более высокие уровни, например, при выполнении системного вызова, который может включать в себя диспетчеризацию потока, диспетчеризацию памяти или ввод-вывод. Если поток поднимает IRQL на уровень dispatch или еще выше, на его процессоре не

будет больше происходить ничего, относящегося к планированию потоков, пока уровень IRQL не будет опущен ниже уровня dispatch. Поток выполняется на dispatch-уровне и выше, блокирует активность планировщика потоков и мешает контекстному переключению на своем процессоре.

Поток, запущенный в режиме ядра, может быть запущен на APC-уровне, если он запускает специальный APC-вызов ядра, или он может временно поднять IRQL до APC-уровня, чтобы заблокировать доставку специальных APC-вызовов ядра (см. главу 3). Но выполнение на APC-уровне не изменяет поведение потока, связанного с планированием, по сравнению с другими потоками; влияние оказывается только на доставку потоку APC-вызовов ядра. Фактически поток, выполняемый в режиме ядра на APC-уровне, может быть прерван в пользу потока с более высоким приоритетом, запущенным в пользовательском режиме на уровне passive.

Использование инструментальных средств для работы с уровнями приоритета

Изменить и просмотреть базовый приоритет процесса можно с помощью Диспетчера задач и Process Explorer. Отдельные потоки в процессе можно уничтожить с помощью Process Explorer, но делать это нужно с большой осторожностью.

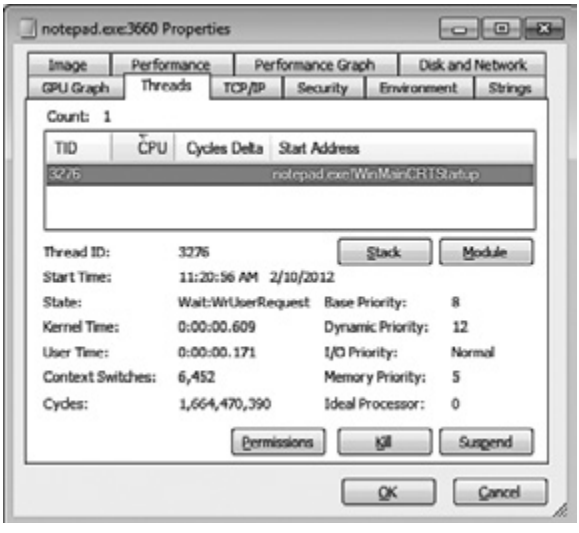
Приоритеты отдельных потоков можно просмотреть с помощью таких средств, как Монитор производительности (Performance Monitor), Process Explorer или WinDbg. Хотя от повышения или снижения приоритета процесса и может быть какая-то польза, обычно нет никакого смысла настраивать приоритеты отдельных потоков внутри процесса, потому что только тот, кто полностью разбирается в программе (разработчик), поймет относительную важность потоков внутри процесса.

Единственный способ определения для процесса стартового класса приоритета заключается в использовании команды start в окне командной строки Windows. Если нужно, чтобы программа каждый раз стартовала с определенным приоритетом, можно определить ярлык, используемый для команды start, начиная его командную строку с команды cmd /c. Эта команда запустит окно командной строки, выполнит команду в командной строке и завершит работу окна командной строки. Например, чтобы запустить Блокнот (Notepad) с приоритетом процесса low, ярлык должен включать команду cmd /c start /low Notepad.exe.

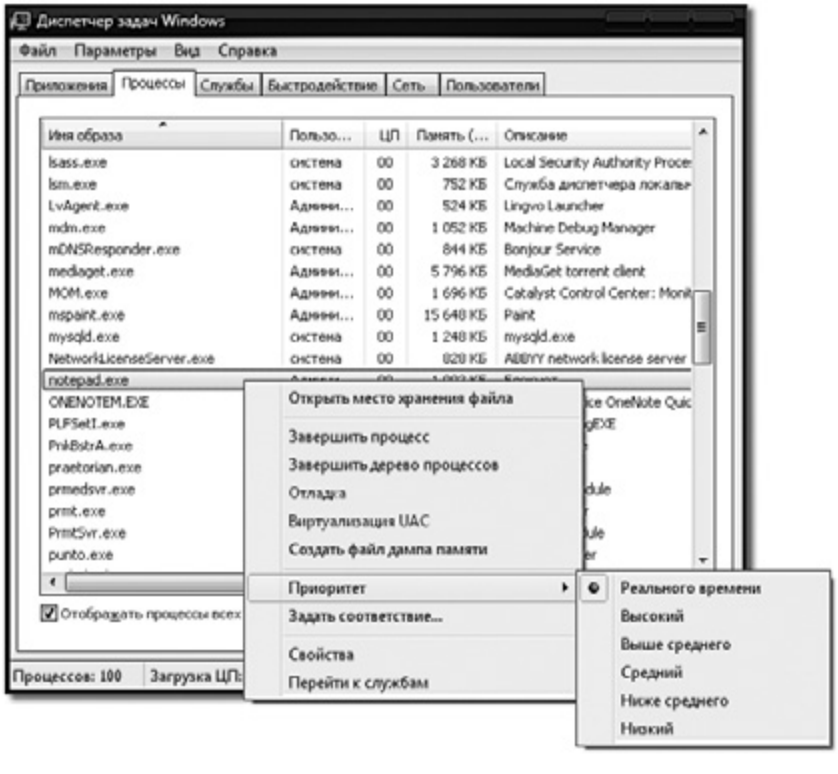
ЭКСПЕРИМЕНТ: ИЗУЧЕНИЕ И ОПРЕДЕЛЕНИЕ ПРИОРИТЕТОВ ПРОЦЕССА JS И ПОТОКОВ

Выполните следующие действия:

1. В окне командной строки с повышенными привилегиями наберите start /realtime notepad. Должно открыться окно программы Блокнот (Notepad).
2. Запустите Process Explorer и выберите Notepad.exe из списка процессов. Дважды щелкните на Notepad.exe, чтобы появилось окно свойств процесса, а затем щелкните на вкладке Threads (Потоки), как показано ниже. Обратите внимание на то, что динамический приоритет потока в процессе Notepad равен 24. Это соответствует значению приоритета реального времени, показанному на следующем изображении.



3. Диспетчер задач может показать вам аналогичную информацию. Нажмите Ctrl+Shift+Esc, чтобы запустить Диспетчер задач, и щелкните на вкладке Процессы (Processes). Щелкните правой кнопкой мыши на имени процесса Notepad.exe и выберите пункт Приоритет (Set Priority). Как показано в следующем диалоговом окне, для Notepad установлен класс процесса Реального времени (Realtime).



ДИСПЕТЧЕР СИСТЕМНЫХ РЕСУРСОВ WINDOWS

В Windows Server 2008 R2 Enterprise Edition и в Windows Server 2008 R2 Datacenter Edition включен дополнительно устанавливаемый компонент, который называется диспетчером системных ресурсов Windows — Windows System Resource Manager (WSRM). Он разрешает администратору настраивать политики, определяющие для процессов инициализацию центрального процессора, настройки родственности и лимиты памяти (как физической, так и виртуальной). Кроме того, WSRM может генерировать отчеты об использовании ресурсов, которые могут использоваться для учета и проверки соглашений об уровне обслуживания пользователей.

Политики могут применяться к определенным приложениям (путем поиска соответствия имени образа с определенными аргументами командной строки или без таких аргументов), пользователям или группам. Эти политики могут планироваться для применения в определенные периоды времени или могут быть включены постоянно.

После установки политики распределения ресурсов для управления определенными процессами служба WSRM отслеживает потребление ресурсов центрального процессора со стороны управляемых процессов и настраивает базовые приоритеты процесса, когда такие процессы не соблюдают целевых выделений времени центрального процессора.

При накладывании ограничений на физическую память используется функция SetProcessWorkingSetSizeEx, с помощью которой устанавливается жестко заданный максимум рабочего набора. Накладывание ограничений на виртуальную память реализовано путем проверки службой закрытой виртуальной памяти, потребляемой процессами. Если эти лимиты превышены, служба WSRM может быть настроена либо на уничтожение процессов, либо на внесение записи в журнал событий. Это поведение может использоваться для определения процессов с утечками памяти, пока они не растратят всю доступную выделяемую память системы. Следует заметить, что лимиты памяти WSRM не применяются к памяти Address Windowing Extensions (AWE), памяти больших страниц или памяти ядра (невыгружаемого или выгружаемого пула).

Состояния потоков

Перед тем как приступить к изучению алгоритмов планирования потоков, нужно разобраться с различными состояниями выполнения, в которых может находиться поток. У потока могут быть следующие состояния:

- **Готов (Ready).** Поток находится в состоянии готовности, ожидая выполнения (или в готовности возвращению в память после завершения ожидания). При поиске потока для выполнения диспетчер рассматривает только пул потоков, находящихся в состоянии готовности.
- **Готов, но отложен (Deferred ready).** Это состояние используется для потоков, выбранных на выполнение на конкретном процессоре, но еще не запущенных на нем. Наличие этого состояния обусловлено тем, что ядро может свести к минимуму период времени блокировки каждого процессора при планировании обращения к удерживаемой базе данных.

- **В повышенной готовности (Standby).** Поток в состоянии повышенной готовности был выбран для запуска следующим на конкретном процессоре. Как только сложатся соответствующие условия, диспетчер выполняет контекстное переключение на этот поток. Для каждого процессора в системе в состоянии повышенной готовности может быть только один поток. Следует заметить, что поток может быть вытеснен из состояния повышенной готовности даже до начала его выполнения, если, к примеру, до того, как начнется выполнение потока, находящегося в повышенной готовности, станет готов к работе поток с более высоким приоритетом.
- **Выполняется (Running).** Как только диспетчер выполняет контекстное переключение на поток, тот входит в состояние выполнения. Его выполнение продолжается до тех пор, пока не истечет его квант времени (и не будет готов другой поток с тем же приоритетом), он не будет вытеснен потоком с более высоким приоритетом, он не завершит свою работу, он не уступить выполнение или он самостоятельно не перейдет в состояние ожидания.
- **Ожидает (Waiting).** Поток может войти в состояние ожидания в нескольких случаях: поток может самостоятельно ожидать объекта для синхронизации своего выполнения, операционная система может ждать от имени потока (например, для разрешения страничного ввода-вывода), или подсистема окружения может приказать потоку приостановиться. Когда ожидание потока заканчивается, то в зависимости от приоритета поток либо немедленно начинает выполняться, либо возвращается в состояние готовности.
- **В переходном состоянии (Transition).** Поток входит в переходное состояние, если он готов к выполнению, но его стек ядра выгружен из памяти. Как только его стек ядра вернется в память, поток войдет в состояние готовности.
- **Завершен (Terminated).** Когда поток заканчивает выполнение, он входит в состояние завершения. Как только поток будет завершен, объект потока исполняющей системы (структура данных с описанием потока в невыгружаемом пуле) может быть освобожден или не освобожден (политику, определяющую время удаления объекта, устанавливает диспетчер объектов).
- **Инициализирован (Initialized).** Это состояние используется внутри системы при создании потока.

В табл. 5.4 дается описание переходов между состояниями потоков, а на рис. 5.16 показана упрощенная версия. (Показанные числа являются показаниями счетчика производительности состояния потока.) В упрощенной версии состояния Ready, Standby и Deferred Ready представлены вместе. Это отражает тот факт, что состояния Standby и Deferred Ready действуют в качестве временных заполнителей для процедур планирования. Эти состояния почти всегда имеют весьма небольшую продолжительность, а потоки в этих состояниях всегда быстро переходят в состояния Ready, Running или Waiting. Подробности происходящего при каждом переходе будут рассмотрены чуть позже.

Таблица 5.4. Состояния потока и переходы между ними

	Init	Ready	Running	Standby	Terminated	Waiting	Transition	Deferred Ready	
Init									Поток становится инициализированным в ходе первых нескольких моментов его создания (KeStartThread)
Ready								Поток добавляется в базу данных готовых потоков диспетчера идеального процессора	
Running		Выбран функцией KiSearchForNewThread		Выбран для выполнения локальным центральным процессором		Вытеснение после удовлетворения ожидания			
Standby		Выбран функцией KiSelectNextThread						Выбран функцией Ki-DeferredReady-Thread для удаленного центрального процессора	

Terminated	Уничтожен до завершения работы функции Psp-InsertThread		Уничтожен						Поток может уничтожить только сам себя. Он должен быть в состоянии Running до входа в функцию KeTerminateThread
Waiting			Поток входит в ожидание						Ждать могут только запущенные потоки
Transition						Стек ядра уже не находится в резидентном состоянии			В состояние transition могут переходить только ожидающие потоки
Deferred Ready	Последний шаг в функции Psp-InsertThread	Родственность изменяется	Поток становится вытесненным (если старый процессор больше не доступен)	Родственность изменяется		Ожидание удовлетворения (но не вытеснения)	Загрузка стека ядра завершена		

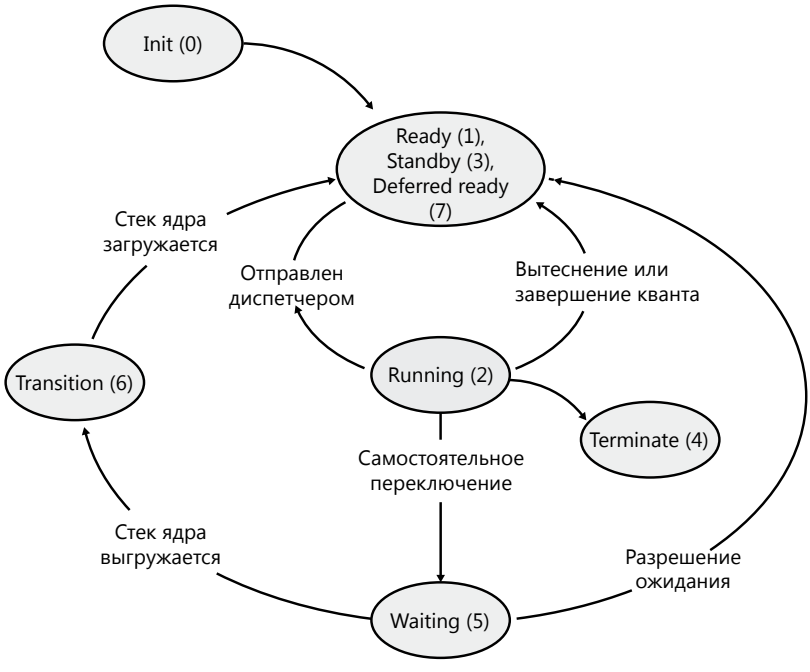


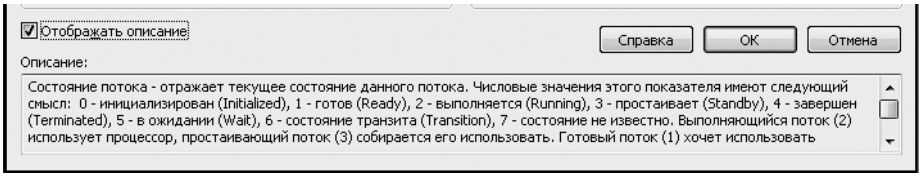
Рис. 5.16. Упрощенная версия состояний потока и переходов между ними

ЭКСПЕРИМЕНТ: ИЗМЕНЕНИЕ СОСТОЯНИЙ ПЛАНИРОВАНИЯ ПОТОКА

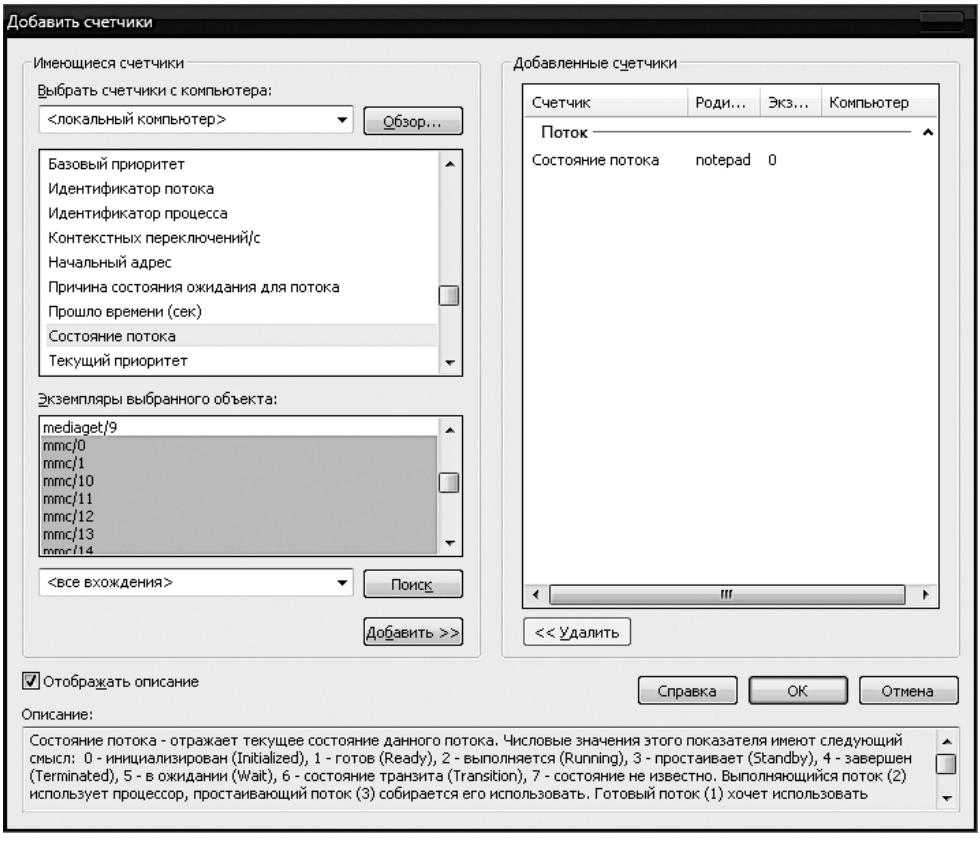
Изменения состояний планирования потока можно изменить в Windows с помощью оснастки Производительность (Performance). Это средство может пригодиться при отладке многопотокового приложения, когда нет сведений о состоянии потоков, запущенных в процессе. Для наблюдения изменений состояния планирования потока с помощью оснастки Производительность (Performance) выполните следующие действия:

1. Запустите Блокнот (Notepad.exe).
2. Запустите оснастку Производительность (Performance), выбрав из меню Пуск (Start) пункты Все программы (All Programs), Администрирование (Administrative Tools), Системный монитор (Performance Monitor). Щелкните на пунктах Средства наблюдения (Monitoring Tools), Системный монитор (Performance Monitor).
3. Выберите режим просмотра графика, если включен какой-нибудь другой режим просмотра.
4. Щелкните правой кнопкой мыши на графике и выберите пункт Свойства (Properties).
5. Щелкните на вкладке График (Graph) и измените значение поля Диапазон значений вертикальной шкалы Максимум (chart vertical scale maximum) на 7. (Из пояснительного текста о счетчике производительности будет видно, что состояния потока пронумерованы от 0 до 7.) Щелкните на кнопке OK.
6. Щелкните на панели инструментов на кнопке Добавить (Add), чтобы появилось диалоговое окно Добавить счетчики (Add Counters).

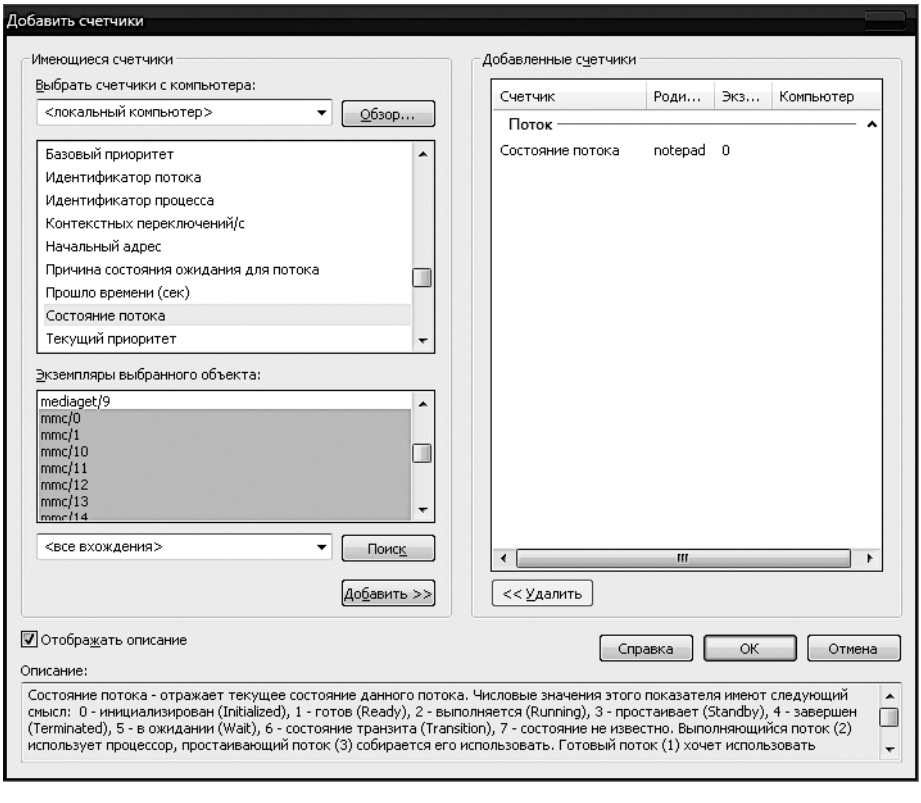
7. Выберите объект подсчета производительности Поток (Thread), а затем выберите счетчик Состояние потока (Thread State). Установите флажок Отображать описание (Show Description), чтобы видеть определения значений.



8. В поле Экземпляры выбранного объекта (Instances) выберите <все вхождения> (<All instances>) и наберите Notepad перед щелчком на кнопке Поиск (Search). Прокрутите список вниз, пока не увидите процесс Блокнота (notepad/0); выберите его и щелкните на кнопке Добавить (Add).
9. Прокрутите назад список, показанный в поле Экземпляры выбранного объекта (Instances), до появления в нем процесса Mmc (процесса консоли управления Microsoft Management Console, запустившей Системный монитор), Выберите все потоки (mmc/0, mmc/1 и т. д.), а затем добавьте их к графику, щелкнув на кнопке Добавить (Add). Перед этим щелчком вы должны увидеть в диалоговом окне примерно следующее.



10. Теперь закройте диалоговое окно Добавить счетчики (Add Counters), щелкнув на кнопке OK.
11. Вы должны увидеть состояние потока Notepad (самая верхняя линия на следующем рисунке), выражающееся числом 5. Как показано в пояснительном тексте, который вы видели в описании действия 7, это число представляет собой состояние ожидания (потому что поток ждет GUI-ввода).
12. Обратите внимание на то, что один поток в процессе Mmc (запустивший оснастку Производительность) находится в состоянии выполнения (число 2). Этот тот самый поток, который запрашивает состояния потока, поэтому он всегда отображается в состоянии выполнения.
13. Вы никогда не увидите Notepad в состоянии выполнения (если только вы не работаете на мультипроцессорной системе), потому что Mmc всегда находится в состоянии выполнения, когда он собирает состояния отслеживаемых потоков.



База данных диспетчера

Чтобы принимать решения по планированию потоков, ядро поддерживает набор структур данных, собирательно известный как база данных диспетчера (рис. 5.17). В базе данных диспетчера отслеживается, какой поток ожидает выполнения и на каких процессорах какие потоки выполняются.

Как показано на рис. 5.17, для улучшения масштабируемости, включая параллельную диспетчеризацию потоков, на мультипроцессорных системах Windows у каждого процессора есть для диспетчера очередь готовых потоков. Таким образом, каждый центральный процессор может проверять свои собственные очереди готовых потоков, чтобы запускать следующий поток, не блокируя общесистемные очереди готовых потоков.

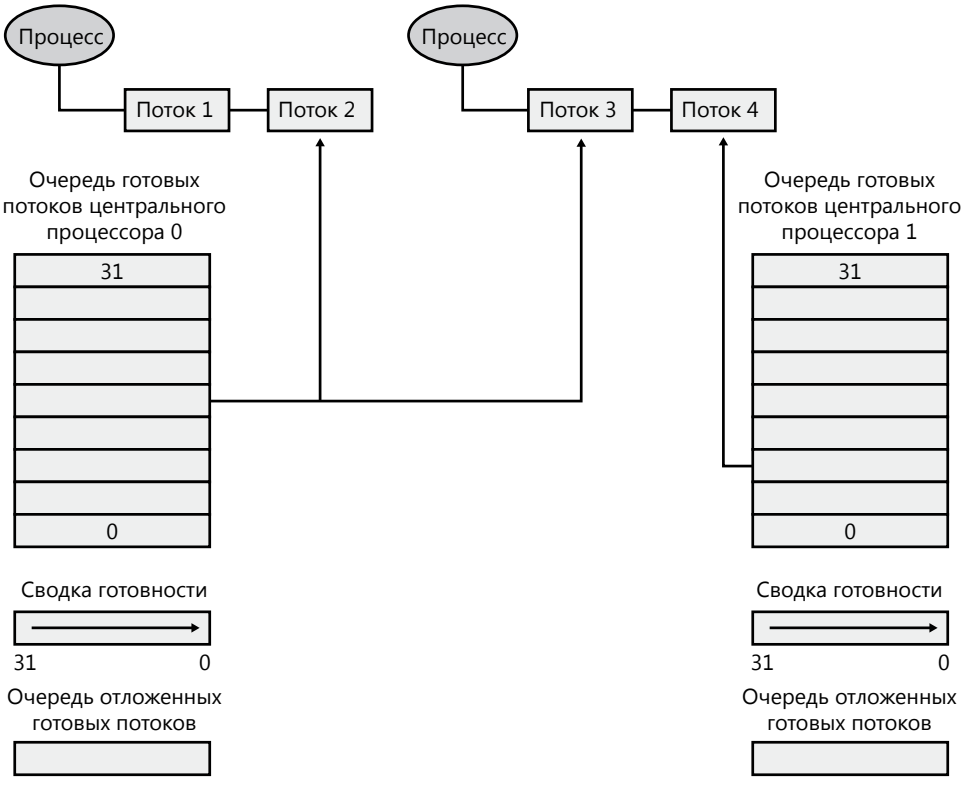


Рис. 5.17. База данных диспетчера мультипроцессорной Windows

Очереди готовых потоков для каждого процессора, а также суммарные данные о готовых потоках являются частью структуры блока управления процессора — processor control block (PRCB). (Чтобы увидеть поля в PRCB, наберите в отладчике ядра команду dt nt!_kprcb.) Имя каждого выделенного далее компонента является полем, составляющим структуру PRCB.

Используемые диспетчером очереди готовых потоков (DispatcherReadyListHead) содержат потоки, находящиеся в состоянии готовности в ожидании планирования их выполнения. Для каждого из 32 уровней приоритета используется по одной очереди. Чтобы ускорить выбор каждого потока на запуск или вытеснение, Windows поддерживает 32-битную двоичную маску, которая называется сводкой готовности (ReadySummary). Каждый установленный бит служит признаком одного или нескольких потоков в очереди готовых потоков для того или иного уровня приоритета. (Нулевой разряд представляет приоритет 0 и т. д.)

Вместо сканирования каждого списка готовности, чтобы узнать, пуст он или нет (чтобы принимать решения по планированию в зависимости от количества потоков разного уровня приоритета), обычной командой процессора сканируется всего один бит и определяется наивысший набор битов. Независимо от количества потоков в очереди их готовности, на эту операцию уходит постоянное количество времени, поэтому иногда можно увидеть, что на алгоритм планирования Windows ссылаются как на O(1) или алгоритм постоянного времени.

База данных диспетчера синхронизируется путем повышения IRQL на DISPATCH_LEVEL (см. главу 3, раздел «Диспетчеризация системных прерываний»). Повышение IRQL таким образом не позволяет другим потокам прерывать диспетчеризацию потоков на процессоре, потому что потоки обычно запускаются с уровнем запроса прерываний IRQL 0 или 1. Но требования одним повышением IRQL не ограничиваются, поскольку другие процессоры могут одновременно поднять уровень до такого же значения IRQL и предпринять попытку работы с базой данных их диспетчера (см. раздел «Мультипроцессорные системы»).

Кванты времени

Как уже ранее упоминалось, квант представляет собой количество времени, получаемое потоком на выполнение, до того как Windows не проверит наличие другого потока с таким же уровнем приоритета, ожидающего запуска. Если поток исчерпал свой квант, а других потоков с его уровнем приоритета нет, Windows позволяет потоку выполняться в течение еще одного кванта времени.

На клиентских версиях Windows потоки по умолчанию выполняются в течение двух интервалов таймера (clock intervals), а на серверных системах по умолчанию поток выполняется в течение 12 интервалов таймера. Причина более продолжительного значения по умолчанию на серверных системах заключается в стремлении минимизировать контекстные переключения. За счет более продолжительного кванта времени серверные приложения, пробуждаемые в результате клиентского запроса, имеют более высокий шанс на завершение обработки запроса и возвращение в состояние ожидания до окончания их кванта времени.

Продолжительность интервала таймера варьируется в зависимости от аппаратной платформы. Частота прерываний таймера зависит от HAL, а не от ядра. Например, интервал таймера на большинстве однопроцессорных систем x86 (они больше не поддерживаются Windows и упоминаются здесь только для примера) составляет около 10 миллисекунд, на большинстве мультипроцессорных систем x86 и x64 он составляет порядка 15 миллисекунд. Этот интервал таймера хранится в переменной ядра KeMaximumIncrement в виде сотен наносекунд.

Поскольку время выполнения потока вычисляется на основе циклов процессора, хотя потоки по-прежнему запускаются в единицах времени, измеряемых интервалами таймера, система не пользуется подсчетом тактовых импульсов в качестве решающего фактора продолжительности выполнения потока и истечения его кванта времени. Вместо этого, когда система запускается, делается вычисление, результатом которого является количество тактовых циклов, которому равен каждый квант времени (значение сохраняется в переменной ядра KiCyclesPerClockQuantum). Это вычисление делается путем умножения тактовой частоты процессора в герцах (числа тактовых импульсов центрального процес-

сора в секунду) на количество секунд, затрачиваемое на запуск одного такта системных часов (на основе KeMaximumIncrement).

Результат такого метода вычисления состоит в том, что потоки на самом деле запускаются не на количество квантов времени, основанное на тактах системных часов, а на время, определенное квантовой целью, которая представляет собой приблизительный подсчет количества тактовых импульсов центрального процессора, потребленное потоком до того, как он уступит свою очередь другому потоку.

Эта квантовая цель должна быть равна количеству тактов интервального таймера, поскольку, как вы только что увидели, подсчет тактовых импульсов на квант основан на величине интервала системного таймера, которую можно проверить с помощью следующего эксперимента. С другой стороны, поскольку потоку назначаются не циклы прерываний, фактический интервал таймера может быть длиннее.

ЭКСПЕРИМЕНТ: ОПРЕДЕЛЕНИЕ ВЕЛИЧИНЫ ИНТЕРВАЛА СИСТЕМНОГО ТАЙМЕРА

Интервал системного таймера возвращает Windows-функция GetSystemTimeAdjustment. Для определения величины этого интервала нужно загрузить и запустить программу Clockres из инструментария Windows Sysinternals (www.microsoft.com/technet/sysinternals). Вывод, полученный на двухъядерной 64-разрядной системе Windows 7, имеет следующий вид:

```
C:\>clockres

ClockRes v2.0 - View the system clock resolution
Copyright (C) 2009 Mark Russinovich
SysInternals - www.sysinternals.com

Maximum timer interval: 15.600 ms
Minimum timer interval: 0.500 ms
Current timer interval: 15.600 ms
```

■

Учет квантового времени

У каждого процесса в его блоке управления (KPROCESS) есть значение перезапуска кванта. Это значение используется при создании новых потоков внутри процесса и дублируется в блок управления потоком (KTHREAD), после чего используется при назначении потоку новой квантовой цели. Значение перезапуска кванта при хранении измеряется в фактических квантовых единицах (что это означает, мы вскоре узнаем), которые затем умножаются на количество тактовых импульсов на квант времени, в результате чего получается квантовая цель.

По мере выполнения потока, тактовые циклы центрального процессора тратятся на различные события — переключения контекста, прерывания и конкретные решения по планированию. При возникновении прерывания от интервального таймера количество затраченных тактовых циклов центрального процессора достигает (или уже превышает) квантовую цель, квант истекает и запускается обработка. Если есть еще один поток, ожидающий запуска и имеющий такой же приоритет, происходит переключение контекста на следующий поток в очереди готовых потоков.

Внутри системы квантовая единица представлена в виде одной трети от такта интервального таймера. (Следовательно, один такт равен трем квантовым единицам.) Это означает, что на клиентских системах Windows потоки по умолчанию имеют значение перезапуска кванта равное 6 (2 × 3) и что серверные системы имеют значение перезапуска кванта равное 36 (12 × 3). Поэтому значение переменной KiCyclesPerClockQuantum в конце ранее рассмотренных вычислений делится на три, поскольку исходное значение описывает только количество тактовых циклов центрального процессора, приходящееся на один такт интервального таймера.

Причина, по которой квант был сохранен внутри системы в виде доли такта таймера, а не в виде целого такта, состояла в том, чтобы позволить частичный расход кванта на завершение ожидания в версиях Windows, предшествовавших Windows Vista. В предыдущих версиях интервальный таймер использовался для истечения кванта времени. Если бы не была сделана эта поправка, потоки могли бы иметь никогда не снижаемый квант времени. Например, если поток выполняется, входит в состояние ожидания, опять выполняется и входит еще в одно состояние ожидания, но при этом он никогда не был текущим выполняемым потоком при запуске интервального таймера, то его квант никогда не будет затрачен на то время, в течение которого он выполнялся. Поскольку теперь у потоков тратятся не кванты, а тактовые циклы центрального процессора и поскольку этот процесс уже не зависит от интервального таймера, такие поправки уже не нужны.

ЭКСПЕРИМЕНТ: ОПРЕДЕЛЕНИЕ КОЛИЧЕСТВА ТАКТОВЫХ ЦИКЛОВ НА ОДИН КВАНТ

В Windows нет какой-либо функции, показывающей количество тактовых циклов на один квант, но на основе предоставленных вычислений и описаний это количество можно определить самостоятельно, выполняя следующие действия и используя такой отладчик ядра, как WinDbg в режиме локальной отладки:

1. Возьмите тактовую частоту вашего процессора, определяемую системой Windows. Можно воспользоваться значением, хранящимся в PRCB-поле MHz, которое можно вывести с помощью команды !cpuinfo. На двухъядерной системе Intel, работающей с тактовой частотой 2829 МГц, этот вывод будет иметь следующий вид:

```
lkd> !cpuinfo
CP  F/M/S Manufacturer  MHz PRCB Signature    MSR 8B Signature Features
 0  6,15,6 GenuineIntel 2829 000000c700000000 >000000c700000000<a00f3fff
 1  6,15,6 GenuineIntel 2829 000000c700000000                    a00f3fff
                     Cached Update Signature 000000c700000000
                    Initial Update Signature 000000c700000000
```

2. Переведите полученное число в герцы (Гц). Получится число тактовых циклов центрального процессора, имеющее место на вашей системе каждую секунду. В данном случае это будет 2 829 000 000 циклов в секунду.
3. Возьмите с помощью программы clockres значение интервала таймера на вашей системе. Им оценивается продолжительность периода времени между запусками таймера. На системе, взятой в качестве примера, этот интервал был равен 15,600100 мс.

4. Преобразуйте это число в количество запусков интервального таймера в секунду. В одной секунде 1000 мс, поэтому разделите число, полученное при выполнении действия 3, на 1000. В данном случае таймер запускается каждые 0,0156001 секунды.
5. Умножьте полученный результат на количество циклов в секунду, полученное при выполнении действия 2. В данном случае на каждый интервал таймера приходится 44 132 682,9 цикла.
6. Вспомните, что каждая квантовая единица равна одной трети интервала таймера, поэтому разделите количество циклов на три. В нашем примере получится 14 710 894, или 0xE0786E в шестнадцатеричном виде. Это количество тактовых циклов, которое должно приходиться на каждую квантовую единицу на системе, запущенной с тактовой частотой 2829 МГц и с интервалом таймера, равным приблизительно 15 мс.
7. Чтобы проверить свои вычисления, выведите дамп значения переменной KiCyclesPerClockQuantum на своей системе. Это значение должно совпадать с полученным результатом.

```
lkd> dd nt!KiCyclesPerClockQuantum L1
    81d31ae8  00e0786e
```

■

Управление величиной кванта

Квант потока можно изменить для всех процессов, но вы можете выбрать только одну из двух настроек: короткую (2 такта таймера, значение, использующееся по умолчанию для клиентских машин) или длинную (12 тактов таймера, значение, использующееся по умолчанию для серверных систем).

ПРИМЕЧАНИЕ

Используя объект задания на системе, работающей с длинными квантами, вы можете выбирать для процесса в задании другие значения кванта. Дополнительные сведения об объекте задания даны в разделе «Объекты заданий».

Чтобы изменить эту настройку, щелкните правой кнопкой мыши на значке Компьютер (Computer) на рабочем столе или выберите в Windows Explorer пункт меню Свойства (Properties), щелкните на пункте Дополнительные параметры системы (Advanced System Settings), перейдите на вкладку Дополнительно (Advanced), щелкните на кнопке Settings (Параметры) в области Быстродействие (Performance) и, наконец, перейдите на вкладку Дополнительно (Advanced). Появляющееся в результате диалоговое окно показано на рис. 5.18.

Выбор Оптимизировать работу: программ (Programs), определяющий использование коротких, переменных квантов, устанавливается по умолчанию для клиентских версий Windows. Если вы установили службы терминалов — Terminal Services — на серверной системе и настроили сервер в качестве сервера приложений, выбирается эта настройка, чтобы у пользователей на терминале сервера были такие же настройки кванта, которые обычно устанавливаются на настольной или клиентской системе. Можно также выбрать эту настройку самостоятельно, если запускать Windows Server в качестве своей настольной операционной системы.

Выбор Оптимизировать работу: служб, работающих в фоновом режиме (Background Services) определяет использование длинных, фиксированных кван-

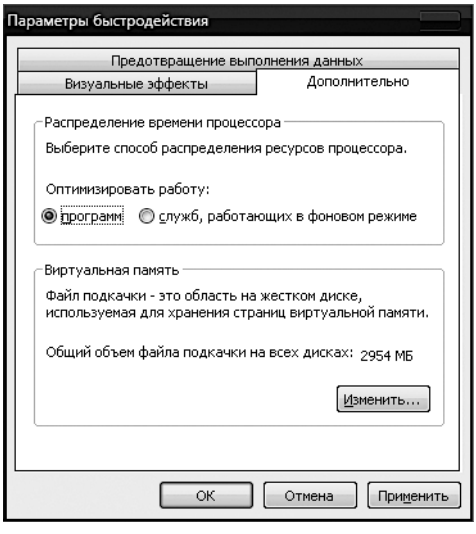


Рис. 5.18. Настройка кванта в диалоговом окне Параметры быстродействия (Performance Options)

тов, и устанавливается по умолчанию для серверных систем. Единственной причиной выбора этой настройки на системе рабочей станции является намерение использовать эту рабочую станцию в качестве серверной системы. Но, поскольку эффект от изменения этой настройки проявляется немедленно, может быть смысл ее использования появится в случае предстоящей работы машины с нагрузкой, напоминающей фоновый режим работы сервера. Например, при длительных вычислениях, кодировании или имитационном моделировании, требующих работы на всю ночь, может быть временно выбран режим Оптимизировать работу: служб, работающих в фоновом режиме (Background Services), а утром работу системы можно вернуть в режим Оптимизировать работу: программ (Programs).

В заключение, поскольку режим Оптимизировать работу: программ (Programs) допускает переменные кванты времени, давайте разберемся, чем управляется их переменчивость.

Переменные кванты

Когда разрешено использование переменных квантов, в таблицу PspForegroundQuantum, которая используется функцией PspComputeQuantum, загружается таблица переменных квантов (PspVariableQuantums). Согласно алгоритму работы этой функции, будет выбран соответствующий индекс кванта на основе того, работает процесс на первом плане или нет (то есть содержится ли в нем поток, владеющий окном первого плана на рабочем столе). Если нет, выбирается индекс, равный нулю, соответствующий рассмотренному ранее кванту потока по умолчанию. Если это процесс первого плана, индекс кванта соответствует разносу приоритетов.

Значение разноса приоритетов определяет повышение приоритета, которое планировщик будет применять к потокам первого плана, и он, таким образом, составит пару с соответствующим расширением квоты: для каждого дополни-

тельного уровня приоритета (вплоть до 2) потоку будет дан еще один квант. Например, если поток получил увеличение приоритета на один уровень, он также получает дополнительный квант. По умолчанию Windows устанавливает для потоков первого плана максимально возможное повышение приоритета, это означает, что разнос приоритетов будет равен 2, поэтому выбор индекса кванта 2 в таблице переменных квантов приведет к тому, что поток получил два дополнительных кванта, что в сумме составит три кванта.

В табл. 5.5 дается описание точных значений квант (вспомним, что они аккумулируются в единицах, представляющих собой одну треть от такта интервального таймера), которые будут выбраны на основе индекса кванта и используемой настройки квантов.

Таблица 5.5. Значения квантов

	Короткий индекс кванта			**Длинный индекс кванта**		
Переменное значение	6	12	18	12	24	36
Фиксированное значение	18	18	18	36	36	36

Таким образом, когда окно помещается на первый план на клиентской системе, все потоки в процессе, содержащем тот поток, который владеет окном первого плана, получают утроенные кванты: потоки в процессе первого плана запускаются с квантом, равным шести тактам интервального таймера, а потоки в других процессах располагают своим клиентским квантом, установленным по умолчанию и равным двум тактам таймера. Таким образом, когда происходит обратное переключение из процесса, интенсивно использующего центральный процессор, новый процесс первого плана получит пропорционально больше времени центрального процессора, потому что когда его потоки запустятся, у них будет более продолжительный период работы, чем у фоновых потоков (опять же при условии, что у потоков как процессов первого плана, так и у фоновых процессов одинаковый приоритет).

Параметр реестра, отображающий настройки кванта

Рассмотренный ранее пользовательский интерфейс для управления настройками кванта изменяет параметр реестра HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl\Win32PrioritySeparation. Кроме определения относительной продолжительности кванта потоков (короткий или длинный), этот параметр реестра также определяет, должны ли использоваться переменные кванты, а также приоритетные разделения (что, как вы увидите, определит индекс кванта, используемый, когда разрешены переменные кванты). Это значение состоит из 6 битов, поделенных на три поля по два бита, показанные на рис. 5.19.

Поля, показанные на рис. 5.19, могут быть определены следующим образом:

- **Короткие или длинные.** Значение 1 определяет длинные кванты, а значение 2 определяет короткие кванты. Установка значения 0 или 3 показывает, что будут использованы установки по умолчанию, применяемые на данной системе (короткие кванты для клиентских систем и длинные кванты для серверных систем).

- **Переменные или фиксированные.** Значение 1 означает включение таблицы переменных квантов, использование которой основано на алгоритме, показанном в разделе «Переменные кванты». Значение 0 или 3 показывает, что будут использованы установки по умолчанию, применяемые на данной системе (переменные кванты на клиентских системах и фиксированные кванты для серверных систем).
- **Разнос приоритетов.** Это поле (значение которого хранится в переменной ядра PsPrioritySeparation) определяет приоритетное разделение (вплоть до 2), как объяснялось в разделе «Переменные кванты».

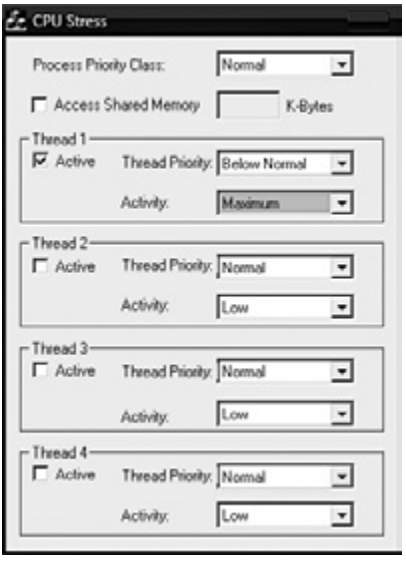


Рис. 5.19. Поля параметра реестра Win32PrioritySeparation

Следует иметь в виду, что при использовании диалогового окна Параметры быстродействия (Performance Options), показанного на рис. 5.18, можно выбрать только одну из двух комбинаций: короткие кванты, с утроенными квантами первого плана, или длинные кванты без изменения квантов для потоков первого плана. Но можно выбрать другие комбинации, непосредственно изменяя параметр реестра Win32PrioritySeparation.

Также следует иметь в виду, что потоки, входящие в процесс, запущенный с классом приоритета простоя (idle), всегда получают единичный квант потока (2 такта таймера), игнорируя любые виды настроек конфигурации квантов, будь то настройки по умолчанию или настройки, произведенные через реестр.

На системах Windows Server, настроенных в виде серверов приложений, исходное значение параметра реестра Win32PrioritySeparation будет равно шестнадцатеричному числу 26, что идентично значению, установленному выбором в диалоговом окне Параметры быстродействия (Performance Options) параметра Оптимизировать работу: программ (Programs). При этом выбирается поведение по определению кванта и повышению приоритета как на клиентских системах Windows, подходящее для сервера, преимущественно использующегося для размещения пользовательских приложений.

На клиентских системах Windows и на серверах, не настроенных в качестве серверов приложений, исходное значение параметра реестра Win32PrioritySeparation будет равно 2. Этим для битовых полей Короткие и длинные (Short vs. Long) и Переменные и фиксированные (Variable vs. Fixed) обусловливаются нулевые значения, которые для этих параметров основываются на поведении системы по умолчанию (в зависимости от того, клиентская это система или серверная). Для поля Разнос приоритетов (Priority Separation) обусловливается значение 2. Как только параметр реестра изменяется с помощью диалогового окна Параметры быстродействия (Performance Options), его исходное значение уже не может быть восстановлено кроме как с помощью непосредственного изменения реестра.

ЭКСПЕРИМЕНТ: ВЛИЯНИЕ ИЗМЕНЕНИЯ НАСТРОЙКИ КВАНТА

Используя локальный отладчик (Kd или WinDbg), можно посмотреть, как две настройки кванта, для программ и для фоновых служб, влияют на таблицы PsPrioritySeparation и PspForegroundQuantum, а также на изменение значения QuantumReset потока в системе. Выполните следующие действия:

1. Откройте оснастку Система (System) в Панели управления (Control Panel) (или щелкните правой кнопкой мыши на значке с именем вашего компьютера на рабочем столе и выберите пункт Свойства (Properties)). Щелкните на пункте Дополнительные параметры системы (Advanced System Settings), щелкните на вкладке Дополнительно (Advanced), щелкните на кнопке Параметры (Settings) в области Быстродействие (Performance) и, наконец, щелкните на вкладке Дополнительно (Advanced). Выберите режим Оптимизировать работу: программ (Programs) и щелкните на кнопке Применить (Apply). Оставьте это окно открытым для продолжения эксперимента.
2. Выведите дамп значений PsPrioritySeparation и PspForegroundQuantum, как это сделано здесь. Ниже показано то, что вы увидите на системе Windows после выполнения действия 1. Обратите внимание на то, как использовалась таблица переменных, коротких квантов, и на то, что к приложениям первого плана будет применено увеличение приоритета, равное 2:

```
lkd> dd PsPrioritySeparation L1
81d3101c  00000002
lkd> db PspForegroundQuantum L3
81d0946c  06 0c 12
...
```

3. Теперь взгляните на значение QuantumReset любого процесса в системе. Как описано ранее, это значение — полный квант каждого потока в системе при его пополнении. Это значение помещается в кэш каждого потока в процессе, но проще будет посмотреть на структуру KPROCESS. Обратите внимание, что в данном случае оно равно 6, поскольку WinDbg, как и большинство других приложений, получает квант, установленный в первой записи таблицы PspForegroundQuantum:

```
lkd> .process
Implicit process is now 85b32d90
lkd> dt nt!_KPROCESS 85b32d90 QuantumReset
nt!_KPROCESS
   +0x061 QuantumReset     : 6 ''
```

4. Теперь в диалоговом окне, открытом при выполнении действия 1, измените настройку в области Быстродействие (Performance), выбрав пункт Оптимизировать работу: служб, работающих в фоновом режиме (Background Services).
5. Повторите команды, показанные в действиях 2 и 3. Вы должны увидеть изменения значений, соответствующие нашему обсуждению данного вопроса в этом разделе:

```
lkd> dd nt!PsPrioritySeparation L1
81d3101c  00000000
lkd> db nt!PspForegroundQuantum L3
81d0946c  24 24 24                                              $$$
lkd> dt nt!_KPROCESS 85b32d90 QuantumReset
nt!_KPROCESS
   +0x061 QuantumReset     : 36 '$'
```

■

Повышение приоритета

Планировщик Windows периодически настраивает текущий приоритет потоков, используя внутренний механизм повышения приоритета. Во многих случаях это делается для уменьшения различных задержек (чтобы потоки быстрее реагировали на события, в ожидании которых они находятся) и повышения их восприимчивости. Другими словами, это повышение применяется для предотвращения сценариев смены приоритетов и зависаний. В данном разделе будет рассмотрен ряд сценариев повышения приоритета, к числу которых относятся следующие сценарии (и их причины):

- Повышение вследствие событий планировщика или диспетчера (сокращение задержек).
- Повышение вследствие завершения ввода-вывода (сокращение задержек).
- Повышение вследствие ввода из пользовательского интерфейса (сокращение задержек и времени отклика).
- Повышение вследствие слишком продолжительного ожидания ресурса исполняющей системы (предотвращение зависания).
- Повышение в случае, когда готовый к запуску поток не был запущен в течение определенного времени (предотвращение зависания и смены приоритетов).

Но, как и любые другие алгоритмы планирования, эти настройки не совершенны, и они могут не принести пользы абсолютно всем приложениям.

ПРИМЕЧАНИЕ

Windows никогда не повышает приоритет потоков в диапазоне реального времени (от 16 до 31). Поэтому планирование относительно других потоков в диапазоне реального времени всегда предсказуемо. Windows предполагает, что при использовании приоритетов потоков реального времени вы знаете, что делаете.

В клиентские версии Windows также включены другие псевдоповышающие механизмы, проявляющие себя при проигрывании мультимедиа. В отличие от других повышений приоритета, эти механизмы применяются непосредственно

в режиме ядра. Повышение приоритета проигрывания мультимедиа обычно управляется службой пользовательского режима, которая называется службой планировщика класса мультимедиа — MultiMedia Class Scheduler Service (MMCSS), но это не является настоящим повышением, служба просто устанавливает по необходимости новые базовые приоритеты для потоков (вызывая для этого исходные API-функции, используемые для изменения приоритетов потоков). Поэтому ни одно из этих правил не касается повышения приоритета. Сначала мы рассмотрим обычные случаи повышения приоритета под управлением ядра, а затем поговорим о MMCSS и о разновидности «повышения», выполняемого этой службой.

Повышение приоритета вследствие событий планировщика или диспетчера

При наступлении события диспетчера вызывается процедура KiExitDispatcher с задачей обработки списка отложенных готовых потоков путем вызова процедуры KiProcessThreadWaitList, а затем вызова процедуры KiCheckForThreadDispatch, чтобы проверить, не должны ли на локальном процессоре быть намечены какие-либо потоки, которые не должны быть спланированы. При каждом наступлении такого события вызывающий код может также указать, какого типа повышение должно быть применено к потоку, а также с каким приращением приоритета должно быть связано это повышение. Следующие сценарии считаются событиями диспетчера AdjustUnwait, потому что они имеют дело с объектом диспетчера, входящим в сигнальное состояние, что может стать причиной пробуждения одного или нескольких потоков:

- В очередь потока поставлен APC-вызов.
- Событие установлено или отправило сигнал.
- Таймер установлен или системное время изменилось, и таймеры должны быть перезапущены.
- Мьютекс был освобожден или ликвидирован.
- Произошел выход из процесса.
- В очередь была вставлена запись или очередь была очищена.
- Был освобожден семафор.
- Поток был приведен в состояние готовности, приостановлен, возобновлен, заморожен или разморожен.
- Первичный UMS-поток ожидает переключения на планируемый UMS-поток.

Для планирования событий, связанных с общедоступным API (например, с SetEvent), применяемое повышение приращения указывается вызывающим кодом. Windows рекомендует разработчикам использовать конкретные значения, которые будут рассмотрены далее. Для приведения в состояние готовности применяется повышение 2, потому что API-функции приведения в состояние готовности не имеют параметра, позволяющего вызывающему коду установить пользовательское приращение.

У планировщика также есть два специальных события диспетчера AdjustBoost, являющиеся частью механизма прав собственности на блокировку приоритета. Эти повышения пытаются исправить ситуации, при которых вызывающий код, владеющий блокировкой на приоритет X, завершает снятие блокировки ожида-

ющего потока с приоритетом <= X. В такой ситуации новый поток владельца должен ждать своей очереди (если он запускается с приоритетом X), или, что еще хуже, он может даже вообще не получить возможности, если его приоритет ниже X. Это влечет за собой то, что освобождаемый поток продолжает выполнение, даже если он должен был разбудить поток нового владельца для получения контроля над процессором. Выдача диспетчером события AdjustBoost вызывается следующими двумя событиями диспетчеризации:

- событием, установленным посредством интерфейса KeSetEventBoostPriority, который используется блокировкой ядра по чтению-записи ERESOURCE;
- шлюзом, установленным посредством интерфейса KeSignalGateBoostPriority, который используется различными внутренними механизмами при освобождении блокировки шлюза.

Повышения приоритета, связанные с завершением ожидания

Повышения приоритета, связанные с завершением ожидания (unwait boosts), пытаются уменьшить время задержки между потоком, пробуждающимся по сигналу объекта (переходя тем самым в состояние Ready), и потоком, фактически приступившим к своему выполнению в процессе, который не находился в состоянии ожидания (переходя тем самым в состояние Running). Поскольку событие, наступление которого ждал поток, может дать информацию того или иного сорта, скажем, о состоянии доступной на данный момент памяти, важно, чтобы это состояние не изменялось закулисно, пока поток все еще находится в состоянии Ready. В противном случае эта информация может стать неактуальной или неверной, как только поток будет запущен.

В разнообразных файлах заголовков Windows указываются рекомендуемые значения, которые должны использоваться кодом режима ядра, вызывающим такие API-функции, как KeSetEvent и KeReleaseSemaphore, и которые соответствуют таким определениям, как MUTANT_INCREMENT и EVENT_INCREMENT. Эти определения всегда устанавливаются в этих заголовках в 1, поэтому вполне можно предположить, что большинство повышений на этих объектах, связанных с завершением ожидания, выльются в повышение, равное 1. В API пользовательского режима приращение указано быть не может, поскольку ни у одного из исходных системных вызовов, например у NtSetEvent, нет параметров для его указания. Вместо этого, когда эти API-функции вызывают основной Ke-интерфейс, они автоматически используют определение _INCREMENT. Также в этом случае из-за изменения системного времени ликвидируются мьютексы и перезапускаются таймеры: система использует повышение по умолчанию, которое обычно будет применяться после того, как мьютекс будет освобожден. И наконец, APC-повышение полностью зависит от вызывающего кода. Скоро будет показано особое использование APC-повышения, связанное с завершением ввода-вывода.

ПРИМЕЧАНИЕ

У некоторых объектов диспетчера нет связанных с ними повышений. Например, при установке или при истечении времени таймера или при подаче сигнала о процессе повышение не применяется.

Все эти повышения на +1 пытаются решить исходную проблему на основе предположения, что как освобождаемый, так и ожидающие потоки запущены с одинаковым приоритетом. Путем повышения приоритета ожидающих потоков на один уровень ожидающий поток должен вытеснить освобождающийся поток, как только операция завершится. К сожалению, на однопроцессорных системах, если это предположение не выполняется, повышение может быть неэффективным: если ожидающий поток ожидает с приоритетом 4, а освобождающийся поток имеет приоритет 8, ожидание с приоритетом 5 не справится с сокращением времени ожидания и принудительным вытеснением. Но на многопроцессорных системах благодаря алгоритмам захвата и балансировки у этого потока с более высоким приоритетом может быть более высокий шанс быть подобранным другим логическим процессором. Этот факт связан с выбором разработчиков, сделанным в исходной NT-архитектуре, в которой владельцы блокировок не отслеживаются (за исключением нескольких блокировок). Это означает, что планировщик не может быть уверен в том, кто на самом деле владеет событием, и на самом ли деле оно будет использоваться в качестве блокировки. Даже если отслеживать владение блокировками, право собственности обычно не передается, чтобы избежать проблем сопровождения, отличающихся от случая с блокировкой ERESOURCE, который будет рассмотрен чуть позже.

Но для некоторых видов блокирующих объектов использование событий и шлюзов в качестве их основных объектов синхронизации, повышение приоритета владельцев блокировки решает дилемму. Кроме того, благодаря схемам распределения процессоров и балансировки нагрузки, которые будут рассмотрены далее, на мультипроцессорной машине готовый поток может быть подобран другим процессором, а его высокий приоритет может повысить шансы на запуск вместо первичного процессора на этом вторичном процессоре.

Повышение приоритета владельца блокировки

Поскольку блокировки ресурсов исполняющей системы (ERESOURCE) и блокировки критических разделов используют основные объекты диспетчеризации, в результате освобождения этих блокировок осуществляются повышения приоритета, связанные с завершением ожидания. С другой стороны, поскольку высокоуровневые реализации этих объектов отслеживают владельца блокировки, ядро может принять более взвешенное решение о том, какого вида повышение должно быть применено с помощью AdjustBoost. В повышениях этого вида значение AdjustIncrement устанавливается на текущий приоритет освобождаемых (или настраиваемых) потоков за минусом любого повышения, связанного с выделением первого плана GUI. Перед тем как будет вызвана функция KiExitDispatcher, кодом события и шлюза вызывается функция KiRemoveBoostThread, чтобы вернуть освобождаемый поток назад к его обычному приоритету (через функцию KiComputeNewPriority). Это действие необходимо, чтобы избежать ситуации сопровождения блокировки (lock convoy), в которой два потока, неоднократно передавая блокировку друг другу, получают все большие повышения приоритета.

Следует заметить, что пуш-блокировки, являющиеся несправедливыми блокировками, потому что владелец блокировки в спорном пути ее приобретения непредсказуем (и, скорее всего, случаен, как и при спин-блокировке), не при-

меняют повышения приоритета из-за владения блокировкой. Причина в том, что это просто содействовало бы вытеснению и быстрому росту приоритета, что не требуется, поскольку блокировка, как только она будет освобождена, тут же становится свободной (пропуская обычный путь ожидания и не ожидания).

Другие отличия между повышением приоритета, связанным с владением блокировкой и повышением приоритета, связанным с завершением ожидания, проявятся в том способе, которым планировщик обычно применяет повышение приоритета, к рассмотрению которого мы перейдем после этого раздела.

Повышение приоритета после завершения ввода-вывода

Windows дает временное повышение приоритета при завершении определенных операций ввода-вывода, при этом потоки, ожидавшие ввода-вывода, имеют больше шансов сразу же запуститься и обработать то, чего они ожидали. Хотя рекомендуемые величины повышения можно найти в заголовочных файлах Windows Driver Kit (WDK) (с помощью поиска строки «#define IO» в файле Wdm.h или в файле Ntddk.h), подходящее значение для увеличения зависит от драйвера устройства. (Эти значения перечислены в табл. 5.6.) Именно драйвер устройства указывает повышение при завершении запроса на ввод-вывод в своем вызове функции ядра IoCompleteRequest. В табл. 5.6 следует обратить внимание на то, что запросы ввода-вывода к устройствам, гарантирующим наилучшую отзывчивость, имеют более высокие значения повышения приоритета.

Таблица 5.6. Рекомендуемые значения повышения приоритета

Устройство	Повышение приоритета
Жесткий диск, привод компакт-дисков, параллельный порт, видеоустройство	1
Сеть, почтовый слот, именованный канал, последовательный порт	2
Клавиатура, мышь	6
Звуковое устройство	8

Как указывалось ранее, это повышение после завершения ввода-вывода надеется на повышения, связанные с завершением ожидания, рассмотренные в предыдущем разделе. А теперь важная деталь состоит в том, что ядро реализует код подачи сигнала в API-функции IoCompleteRequest посредством использования либо APC (для асинхронного ввода-вывода), либо через событие (для синхронного ввода-вывода). Когда драйвер передает в функцию IoCompleteRequest, к примеру, параметр IO_DISK_INCREMENT для асинхронного чтения диска, ядро вызывает KeInsertQueueApc с параметром повышения, установленным на IO_DISK_INCREMENT. В свою очередь, когда ожидание потока разрушается из-за APC, он получает повышение, равное 1.

Следует иметь в виду, что значения повышения, которые были даны в предыдущей таблице, это всего лишь рекомендации от компании Microsoft, разработчики драйверов могут при желании спокойно их проигнорировать, и конкретные специализированные драйверы могут использовать свои собственные значения. Например, драйвер, обрабатывающий ультразвуковые данные от медицинского прибора, ко-

торый должен уведомлять приложение визуализации, работающее в пользовательском режиме о новых данных, будет, наверное, использовать значение повышения равное 8, чтобы получить такое же время ожидания, как и у звуковой карты.

Но во многих случаях из-за особенностей организации стеков драйверов Windows разработчики драйверов часто создают мини-драйверы, обращающиеся к драйверам компании Microsoft, которые снабжают их своим собственным повышением приоритета IoCompleteRequest. Например, разработчики карт контроллеров RAID или SATA будут, как правило, вызывать для завершения обработки их запросов функцию StorPortCompleteRequest. У этого вызова нет никаких параметров для значения повышения, потому что драйвер Storport.sys вставляет подходящее значение при вызове ядра.

Кроме того, в новых версиях Windows при каждом завершении запроса любым драйвером файловой системы (идентифицируемом установкой своего типа устройства в FILE_DEVICE_DISK_FILE_SYSTEM или в FILE_DEVICE_NETWORK_FILE_SYSTEM), всегда вместо этого применяется повышение IO_DISK_INCREMENT, если драйверу передается значение IO_NO_INCREMENT. Таким образом, это значение увеличения становится меньше рекомендованного и больше требуемого ядром.

ПРИМЕЧАНИЕ

Вы можете интуитивно ожидать «более высокой отзывчивости», чем 1 от своей видеокарты или диска, но на самом деле ядро пытается провести оптимизацию в отношении времени ожидания, и некоторые устройства (а также человеческого восприятия) оказываются более чувствительными, чем остальные. Чтобы дать вам представление об этом, примем во внимание, что звуковая карта ожидает данных примерно каждую миллисекунду, чтобы проиграть музыку без заметных сбоев, а видеокарте нужно вывести только 24 кадра в секунду, или один кадр каждые 40 миллисекунд, прежде чем человеческий глаз заметить какие-нибудь сбои.

Повышение при ожидании ресурсов исполняющей системы

Когда поток пытается получить ресурс исполняющей системы (ERESOURCE), который уже находится в исключительном владении другого потока, он должен войти в состояние ожидания до тех пор, пока другой поток не освободит ресурс. Для ограничения риска взаимных исключений исполняющая система выполняет это ожидание, не входя в бесконечное ожидание ресурса, а интервалами по пять секунд.

Если по окончании этих пяти секунд ресурс все еще находится во владении, исполняющая система пытается предотвратить зависание центрального процессора путем завладения блокировкой диспетчера, повышения приоритета потока или потоков, владеющих ресурсом, до значения 14 (если исходный приоритет владельца был меньше, чем у ожидающего, и не был равен 14), перезапуска их квантов и выполнения еще одного ожидания.

Поскольку ресурсы исполняющей системы могут быть либо общими, либо эксклюзивными, ядро сначала повышает приоритет эксклюзивного владельца, а затем проводит проверку общих владельцев и повышает приоритет всех этих владельцев. Когда ожидающий поток опять входит в состояние ожидания, появляется надежда, что планировщик так спланирует работу одного из потоков-владельцев, чтобы у него было достаточно времени для завершения своей

работы и освобождения ресурса. Следует учесть, что этот механизм повышения приоритета используется только в случае отсутствия у ресурса установленного флага запрещения повышения приоритета — Disable Boost, установку которого разработчики могут выбрать при нормальной работе с использованием ресурсов описываемого здесь механизма смены приоритетов.

Следует также заметить, что этот механизм несовершенен. Например, если у ресурса есть несколько общих пользователей, исполняющая система повышает приоритет всех потоков-владельцев до значения 14, что приводит к внезапному всплеску в системе количества высокоприоритетных потоков, обладающих полными квантами времени. Хотя исходный поток-владелец будет запущен первым (поскольку был первым при повышении приоритета и поэтому первым в списке готовых потоков), другие владельцы общего ресурса будут запущены следующими, поскольку приоритет ожидающего потока повышен не был. Только после того, как все владельцы общего ресурса воспользуются возможностью запуститься и их приоритет будет уменьшен до значения ниже приоритета ожидающего потока, этот ожидающий поток наконец-то получит шанс на приобретение ресурса. Поскольку владельцы общего ресурса могут перевести или конвертировать свое владение из общего в эксклюзивное, то, как только эксклюзивный владелец освобождает ресурс, данный механизм может работать не так, как было задумано.

Повышение приоритета потоков первого плана после ожидания

Как в скором времени будет описано, когда поток в процессе первого плана завершает операцию ожидания объекта ядра, ядро повышает его текущий (но не базовый) приоритет на текущее значение переменной PsPrioritySeparation. (Какой процесс отнести к процессам первого плана, решает система управления окнами.) Как было описано в разделе управления квантами времени, в переменной PsPrioritySeparation отражается индекс таблицы квантов, используемый для выбора квантов для потоков первого плана. Но в данном случае значение этой переменной будет использовано в качестве значения повышения приоритета.

Смысл этого повышения заключается в улучшении отзывчивости интерактивных приложений: если дать приложениям первого плана небольшое повышение приоритета при завершении ожидания, у них повышаются шансы сразу же приступить к работе, особенно когда другие процессы с таким же базовым приоритетом могут быть запущены в фоновом режиме.

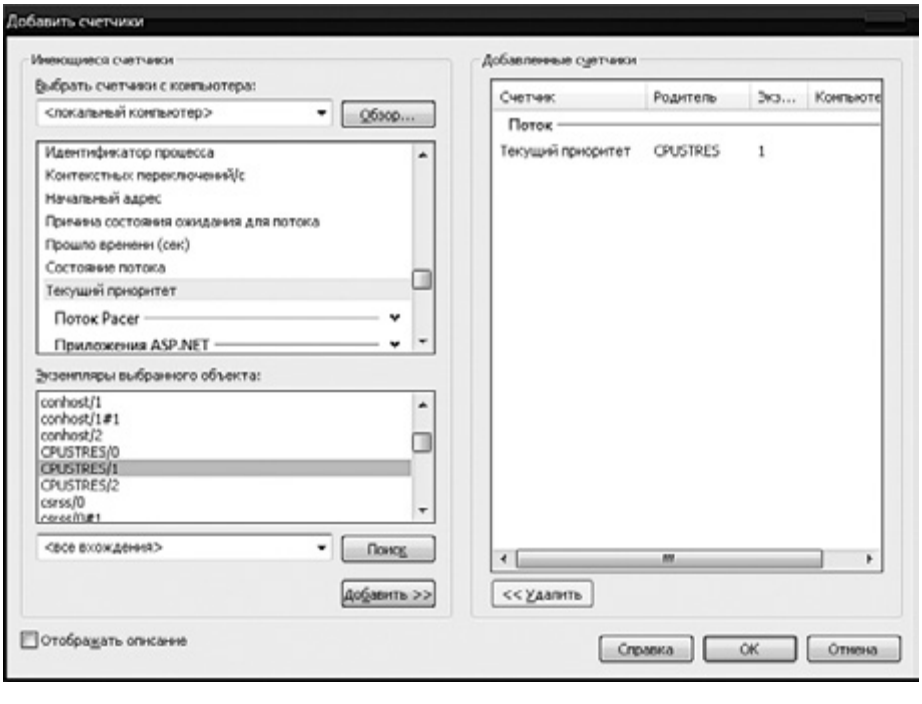
ЭКСПЕРИМЕНТ: НАБЛЮДЕНИЕ ЗА ПОВЫШЕНИЯМИ И СНИЖЕНИЯМИ ПРИОРИТЕТА ПОТОКОВ ПЕРВОГО ПЛАНА

Используя средство CPU Stress (которое можно загрузить с веб-сайта http://live.sysinternals.com/WindowsInternals), можно вести наблюдение за повышением приоритета в действии. Выполните следующие действия:

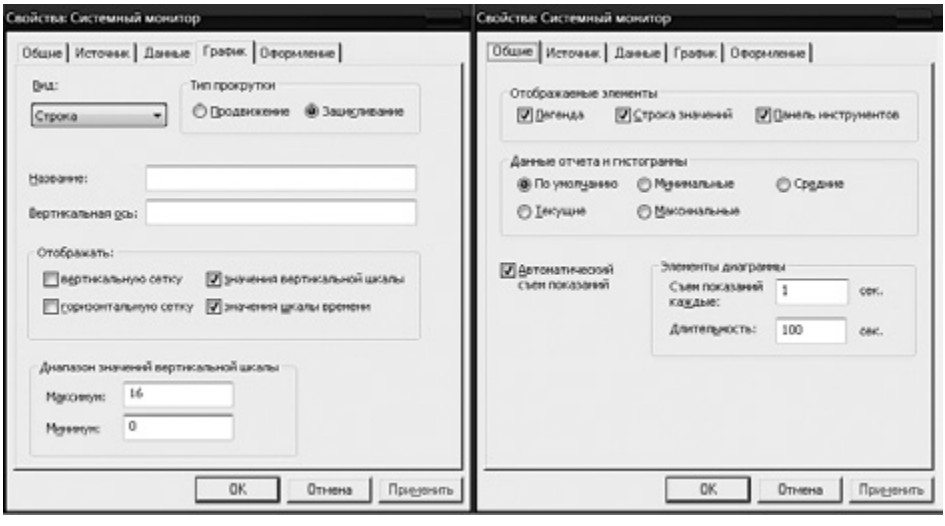
1. Откройте оснастку Система (System) в Панели управления (Control Panel) (или щелкните правой кнопкой мыши на значке с именем вашего компьютера на рабочем столе и выберите пункт Свойства (Properties)). Щелкните на пункте Дополнительные параметры системы (Advanced System Settings), щелкните на вкладке Дополнительно (Advanced), щелкните на кнопке Параметры (Settings) в области Быстродействие (Performance) и, наконец, щелкните на вкладке Дополнительно (Advanced). Выберите режим

Оптимизировать работу: программ (Programs). Это приведет к тому, что значение переменной PsPrioritySeparation станет равным 2.

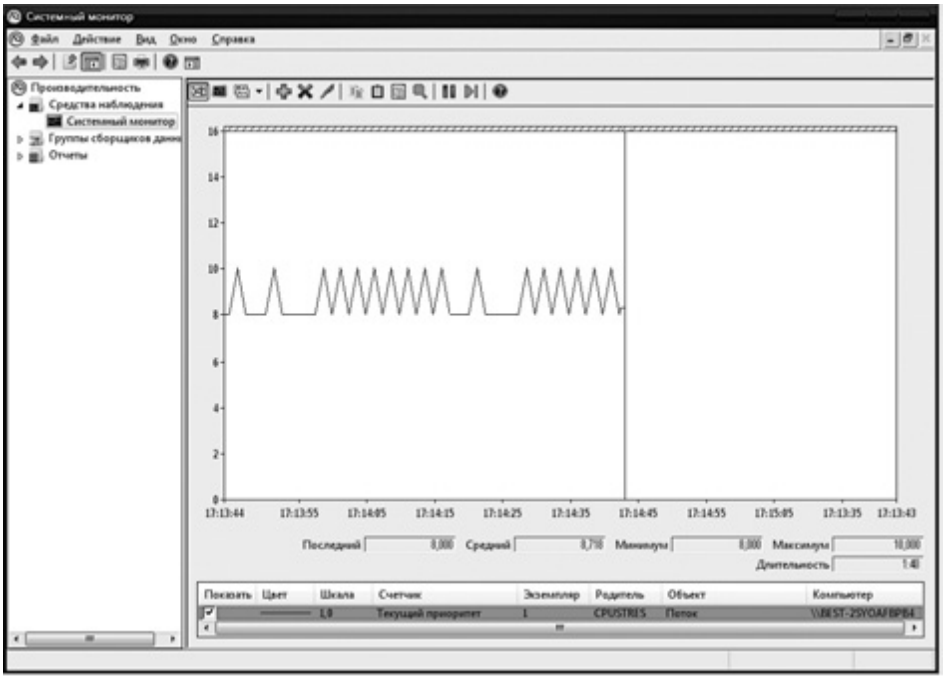
2. Запустите программу Cpustres.exe и измените активность потока 1 (thread 1) с низкой — Low на занят — Busy.
3. Запустите оснастку Производительность (Performance), выбрав из меню Пуск (Start) пункты Все программы (All Programs), Администрирование (Administrative Tools), Системный монитор (Performance Monitor). Щелкните на пунктах Средства наблюдения (Monitoring Tools), Системный монитор (Performance Monitor).
4. Щелкните на кнопке панели инструментов Добавить (Add Counter) (или нажмите клавиши Ctrl+N), чтобы появилось диалоговое окно Добавить счетчики (Add Counters).
5. Выберите объект Поток (Thread), а затем выберите счетчик Текущий приоритет (Priority Current).
6. В поле со списком Экземпляры выбранного объекта (Instances) выберите пункт <все вхождения> (<All Instances>) и щелкните на кнопке Поиск (Search). Прокрутите список, пока не увидите процесс CPUSTRES. Выберите второй поток (поток 1). (Первый поток является потоком GUI.) Вы должны увидеть примерно следующую картину.



7. Щелкните на кнопке Добавить (Add), а затем щелкните на кнопке OK.
8. Из меню Действие (Action) выберите пункт Свойства (Properties). Во вкладке График (Graph) измените настройку Диапазон значений вертикальной шкалы Максимум: (Vertical Scale Maximum) на 16, а во вкладке Общие (General) в области Элементы диаграммы (Graph Elements) установите для поля Съем показаний каждые (Sample Every) интервал 1.



9. Теперь переведем процесс CPUSTRES на первый план. Вы должны увидеть, что приоритет потока CPUSTRES повысился на 2, а затем понизился обратно к значению базового приоритета.



10. Причина, по которой CPUSTRES периодически получает повышения на 2, заключается в том, что отслеживаемый поток находится примерно 25 % времени в спящем режиме, а затем пробуждается. (Это уровень активности Busy.) Повышение применяется при пробуждении потока. Если установить Activity level (Уровень активности) в Maximum (Максималь-

ный), вы не увидите никаких повышений, потому что уровень Maximum в программе CPUSTRES помещает поток в бесконечный цикл. Поэтому поток не вызывает никаких функций ожидания, в результате чего не получает никаких повышений приоритета.

11. Когда эксперимент будет завершен, выйдите из Системного монитора и из CPU Stress. ■

Повышение приоритета после пробуждения GUI-потока

Потоки-владельцы окон получают при пробуждении дополнительное повышение приоритета на 2 из-за активности системы работы с окнами, например поступление сообщений от окна. Система работы с окнами (Win32k.sys) применяет это повышение приоритета, когда вызывает функцию KeSetEvent для установки события, используемого для пробуждения GUI-потока. Смысл этого повышения похож на смысл предыдущего повышения — содействие интерактивным приложениям.

ЭКСПЕРИМЕНТ: НАБЛЮДЕНИЕ ЗА ПОВЫШЕНИЯМИ ПРИОРИТЕТА GUI-ПОТОКОВ

Отслеживая текущий приоритет GUI-приложения и передвигая указатель мыши по окну, можно также отследить повышение приоритета GUI-потоков на 2 со стороны системы работы с окнами при пробуждении этих потоков для обработки сообщений окон. Выполните следующие действия:

1. Откройте оснастку Система (System) в Панели управления (Control Panel) (или щелкните правой кнопкой мыши на значке с именем вашего компьютера на рабочем столе и выберите пункт Свойства (Properties)). Щелкните на пункте Дополнительные параметры системы (Advanced System Settings), щелкните на вкладке Дополнительно (Advanced), щелкните на кнопке Параметры (Settings) в области Быстродействие (Performance) и, наконец, щелкните на вкладке Дополнительно (Advanced). Убедитесь в том, что выбран режим Оптимизировать работу: программ (Programs). Это приведет к тому, что значение переменной PsPrioritySeparation станет равным 2.
2. Запустите из меню Пуск (Start) программу Блокнот, выбрав пункты Все программы (All Programs)▸ Стандартные (Accessories)▸ Блокнот (Notepad).
3. Запустите оснастку Производительность (Performance), выбрав в меню Пуск (Start) пункты Все программы (All Programs)▸ Администрирование (Administrative Tools)▸ Системный монитор (Performance Monitor). Щелкните на пунктах Средства наблюдения (Monitoring Tools)▸ Системный монитор (Performance Monitor).
4. Щелкните на кнопке панели инструментов Добавить (Add Counter) (или нажмите клавиши Ctrl+N), чтобы появилось диалоговое окно Добавить счетчики (Add Counters).
5. Выберите объект Поток (Thread), а затем выберите счетчик Текущий приоритет (Priority Current).
6. В поле со списком Экземпляры выбранного объекта (Instances) наберите Notepad, а затем щелкните на кнопке Поиск (Search). Прокрутите список, пока не увидите запись Notepad/0. Щелкните на ней кнопкой мыши, щелкните на кнопке Добавить (Add), а затем щелкните на кнопке OK.

7. Как и в предыдущем эксперименте, выберите меню Действие (Action), выберите пункт Свойства (Properties). На вкладке График (Graph) измените настройку Диапазон значений вертикальной шкалы Максимум: (Vertical Scale Maximum) на 16, а во вкладке Общие (General) в области Элементы диаграммы (Graph Elements) установите для поля Съем показаний каждые (Sample Every) интервал 1 и щелкните на кнопке OK.
8. Вы увидите приоритет потока 0 в Notepad на уровне 8 или 10. Поскольку Notepad входит в состояние ожидания, а вскоре после того как получил повышение на 2, которое получают потоки процесса первого плана, он еще может не получить снижение с 10 до 8.
9. После вывода на первый план Системного монитора (Performance Monitor) переместите указатель мыши вдоль окна Блокнота (Notepad). (Оба окна должны быть видны на Рабочем столе.) Вы увидите, что по только что рассмотренным причинам приоритет иногда остается на 10 единицах, а иногда на 9. (Причина того, увидеть приоритет Блокнота на уровне 8 вам вряд ли удастся, заключается в том, что работа после повышения приоритета GUI-потока на 2 слишком кратковременна, и уровень приоритета никогда не успевает снижаться более чем на единицу, прежде чем поток не будет снова пробужден благодаря новой активности работы с окном и нового получения повышения приоритета на 2.)
10. Теперь переведите Блокнот на первый план. Вы увидите, что приоритет поднялся до 12 единиц и остался на этом уровне (или упал до 11, потому что может произойти обычное снижение приоритета по окончании кванта), поскольку поток получает два повышения: повышение на 2, применимое к GUI-потокам при пробуждении процесса получающего ввод при работе с окном, и дополнительное повышение на 2, поскольку Блокнот находится на первом плане.
11. Если затем провести указатель мыши над Блокнотом (при том, что он по-прежнему находится на первом плане), можно увидеть, что приоритет упал до 11 единиц (или даже может быть до 10 единиц), поскольку происходит снижение приоритета, которое обычно случается с потоками с повышенным приоритетом при завершении их очереди выполняться. Но применяемое повышение приоритета на 2 единицы обусловлено тем, что процесс остается на первом плане, пока на этом же плане остается само приложение Блокнот.
12. Когда эксперимент будет закончен, выйдите из Системного монитора (Performance Monitor) и из Блокнота (Notepad). ■

Повышения приоритета, связанные с перезагруженностью центрального процессора (CPU Starvation)

Представьте себе следующую ситуацию: имеется выполняемый поток с приоритетом 7, который не дает потоку с приоритетом 4 когда-либо получить процессорное время, но в то же время поток с приоритетом 11 ожидает какой-нибудь ресурс, который заблокирован потоком с приоритетом 4. Но поскольку поток с приоритетом 7 израсходовал только около половины потребляемого времени центрального процесса, процесс с приоритетом 4 никогда не будет выполняться достаточно долго, чтобы завершить свою работу и освободить ресурс, блокирующий поток с приоритетом 11. Что делает Windows в отношении подобной ситуации?

Ранее уже было показано, как код исполняющей системы отвечает за управление ресурсами исполняющей системы при таком развитии событий путем повышения приоритета потоков-владельцев, чтобы у них был шанс на выполнение и освобождение ресурса. Но ресурсы исполняющей системы являются только одной из многих конструкций синхронизации, доступной разработчикам, и технология повышения приоритета к любым другим примитивам применяться не будет. Поэтому в Windows также включен общий механизм ослабления загруженности центрального процессора, который называется диспетчером настройки баланса и является частью потока (речь идет о системном потоке, который существует главным образом для выполнения функций управления памятью).

Один раз в секунду этот поток сканирует очередь готовых потоков в поиске тех из них, которые находятся в состоянии ожидания (то есть не были запущены) около 4 секунд. Если такой поток будет найден, диспетчер настройки баланса повышает его приоритет до 15 единиц и устанавливает квантовую цель эквивалентной тактовой частоте процессора при подсчете 3 квантовых единиц. Как только квант истекает, приоритет потока тут же снижается до обычного базового приоритета. Если поток не был завершен и есть готовый к запуску поток с более высоким уровнем приоритета, поток с пониженным приоритетом возвращается в очередь готовых потоков, где он опять становится подходящим для еще одного повышения приоритета, если будет оставаться в очереди следующие 4 секунды.

Диспетчер настройки баланса на самом деле при своем запуске сканирует не все потоки, находящиеся в состоянии готовности. Для минимизации затрачиваемого на его работу времени центрального процессора он сканирует только 16 готовых потоков; если на данном уровне приоритета имеется больше потоков, он запоминает то место, на котором остановился, и начинает с него при следующем проходе очереди. Кроме того, он за один проход повысит приоритет только 10 потоков, если найдет 10 потоков, заслуживающих именно этого повышения (что свидетельствует о необычно высоко загруженной системе), он прекратит сканирование на этом месте и начнет его с этого же места при следующем проходе.

ПРИМЕЧАНИЕ

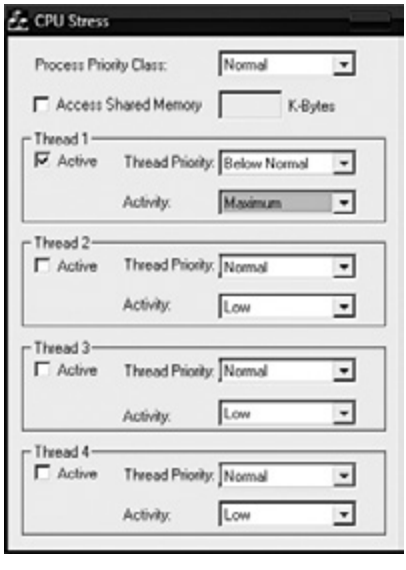
Как уже ранее упоминалось, решения по планированию в Windows не подвержены влиянию со стороны количества потоков и принимаются в фиксированном промежутке времени, или O(1). Поскольку диспетчер настройки баланса нуждается в самостоятельном сканировании очереди готовых потоков, эта операция зависит от количества потоков, имеющихся в системе, и большее количество потоков потребует большего времени сканирования. Но диспетчер настройки баланса не считается частью планировщика или его алгоритмов, это просто расширенный механизм для повышения надежности работы системы. Кроме того, поскольку сканируется только верхушка потоков и очередей, влияние на снижение производительности сведено к минимуму и предсказуемо при худшем развитии событий.

Но сможет ли этот алгоритм всегда решать вопросы инверсии приоритета? Нет, он ни в коем случае не претендует на совершенство. Но со временем потоки, страдающие от перезагруженности центрального процессора, получат достаточно процессорного времени для завершения той обработки, которой они заняты, и снова войдут в состояние ожидания.

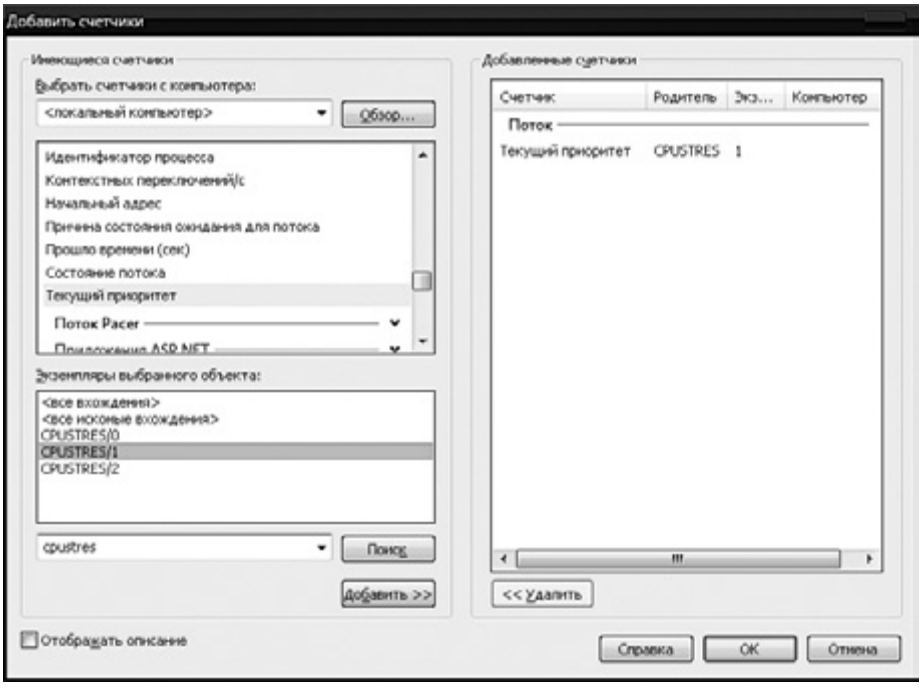
ЭКСПЕРИМЕНТ: НАБЛЮДЕНИЕ ЗА ПРИОРИТЕТОМ ПОТОКОВ ДЛЯ ВЫЯВЛЕНИЯ ПЕРЕЗАГРУЖЕННОСТИ ЦЕНТРАЛЬНОГО ПРОЦЕССОРА

Используя такой инструмент, как CPU Stress, можно наблюдать повышение приоритета в действии. В данном эксперименте вы увидите изменение использования центрального процессора при повышении приоритета потока. Выполните следующие действия:

1. Запустите программу Cpustres.exe. Измените уровень активности активного потока (по умолчанию это Thread 1) с низкого — Low на максимальный — Maximum. Измените приоритет потока с обычного — Normal на ниже обычного — Below Normal. Все это должно выглядеть следующим образом.



2. Запустите оснастку Производительность (Performance), выбрав из меню Пуск (Start) пункты Все программы (All Programs), Администрирование (Administrative Tools), Системный монитор (Performance Monitor). Щелкните на пунктах Средства наблюдения (Monitoring Tools), Системный монитор (Performance Monitor).
3. Щелкните на кнопке панели инструментов Добавить (Add Counter) (или нажмите клавиши Ctrl+N), чтобы появилось диалоговое окно Добавить счетчики (Add Counters).
4. Выберите объект Поток (Thread), а затем выберите счетчик Текущий приоритет (Priority Current).
5. В поле со списком Экземпляры выбранного объекта (Instances) наберите CPUSTRES, а затем щелкните на кнопке Поиск (Search). Прокрутите список, пока не увидите запись, относящуюся ко второму потоку (thread 1). (Первый поток является GUI-потоком.) Окно должно приобрести следующий вид.



6. Щелкните на кнопке Добавить (Add), а затем щелкните на кнопке OK.
7. Поднимите приоритет приложения Системный монитор (Performance Monitor) до реального времени, для чего запустите Диспетчер задач (Task Manager), щелкните на вкладке Процессы (Processes) и выберите процесс Mmc.exe. Щелкните правой кнопкой мыши на названии процесса, выберите пункт Приоритет (Set Priority), а затем выберите пункт Реального времени (Realtime). (Если Диспетчер задач выведет окно предупреждения с сообщением о нестабильности системы, щелкните на кнопке Изменить приоритет (Yes).) Если у вас мультипроцессорная система, нужно будет также поменять родственность процесса: щелкните правой кнопкой мыши и выберите пункт Задать соответствие (Set Affinity). Затем снимите флажки со всех ЦП, за исключением ЦП 0.
8. Запустите еще одну копию CPU Stress. В этой копии измените уровень активности потока Thread 1 с низкого — Low на максимальный — Maximum.
9. Теперь опять переключитесь на Системный монитор (Performance Monitor). Вы увидите активность центрального процессора каждые шесть или около этого секунд, поскольку приоритет потока будет повышаться до 15 единиц. Вы можете заставить проводить обновления несколько чаще, чем каждую секунду, остановив отображение путем нажатия клавиш Ctrl+F, а затем нажав клавиши Ctrl+U, перейдя тем самым на ручной режим обновления показаний счетчиков. Для последующих обновлений продолжайте нажимать клавиши Ctrl+U.

Завершив эксперимент, выйдите из Системного монитора и из двух копий программы CPU Stress. ■

ЭКСПЕРИМЕНТ: «ПРОСЛУШИВАНИЕ» ПОВЫШЕНИЙ ПРИОРИТЕТА

Чтобы «услышать» эффект повышения приоритета, связанного с перезагруженностью центрального процессора, выполните следующие действия на системе, оборудованной звуковой картой:

1. Из-за повышений приоритета, относящихся к MMCSS (которые будут рассмотрены в следующем подразделе), вам нужно остановить службу Multimedia Class Scheduler Service, открыв интерфейс управления службами, перейдя по пунктам меню Пуск (Start), Все программы (Programs), Администрирование (Administrative Tools), Службы и устройства (Services).
2. Запустите Windows Media Player (или какую-нибудь другую программу проигрывания аудиофайлов) и начните проигрывание какого-нибудь аудиоконтента.
3. Запустите утилиту Cpustres и установите уровень активности потока Thread 1 на максимальный — Maximum.
4. Воспользуйтесь Диспетчером задач (Task Manager) для установки родственности как Windows Media Player, так и Cpustres с одним и тем же центральным процессором.
5. Поднимите приоритет потока Thread 1 в Cpustres с обычного — Normal на критичный по времени — Time Critical.
6. Вы услышите, как остановится проигрывание музыки как только связанный с компьютером поток начнет потреблять все доступное время центрального процессора.
7. Время от времени вы услышите небольшие звуковые фрагменты, как только остановленный перезагруженностью процессора поток в процессе проигрывания музыки получит повышение приоритета до 15 единиц и будет выполняться достаточное количество времени, чтобы отправить очередную порцию данных на звуковую карту.
8. Закройте приложения Cpustres и Windows Media Player и снова запустите службу MMCSS. ■

Применение повышений приоритета

Возвращаясь к работе функции KiExitDispatcher, вы увидите, что из нее вызывается функция KiProcessThreadWaitList, чтобы обработать любые потоки в списке отложенных готовых потоков. Именно здесь обрабатывается информация о повышении приоритета, переданная вызывающей процедурой. Это делается путем циклического перебора каждого потока, находящегося в состоянии DeferredReady, разъединяя его блоки ожидания (разъединяются только блоки Active и Bypassed), а затем устанавливая два ключевых значения в находящемся в ядре блоке управления потоком: AdjustReason и AdjustIncrement. Причина заключается в одной из двух возможностей настроек (Adjust), показанных ранее, и в приращении, соответствующем значению повышения. Затем вызывается функция KiDeferredReadyThread, которая превращает поток в готовый к выполнению путем запуска двух алгоритмов: алгоритма выбора кванта и приоритета, который вы сейчас увидите, в двух частях; и алгоритма выбора процессора, который будет показан далее в соответствующем разделе.

Сначала давайте посмотрим на то, когда алгоритм применяет повышение приоритета, что происходит только в тех случаях, когда поток не относится к диапазону приоритетов реального времени.

Что касается повышения приоритета, связанного с завершением ожидания (AdjustUnwait), то оно будет применено, только если поток еще не испытал на себе повышение приоритета сверх обычного и только если поток не был настроен на отключение повышения приоритета путем вызова функции SetThreadPriorityBoost, установившей флаг DisableBoost в структуре данных KTHREAD. Еще одна ситуация, при которой в данном случае может отключиться повышение приоритета, связана с тем, что ядро выяснило, что поток фактически исчерпал свой квант времени (но прерывание от таймера, чтобы зафиксировать этот факт, еще не состоялось), вышел из состояния ожидания, которое длилось менее двух тактов таймера.

Если таких ситуаций в текущий момент не было, новый уровень приоритета потока будет вычислен путем сложения значения AdjustIncrement с текущим базовым приоритетом потока. Кроме того, если известно, что поток является частью процесса первого плана (а это означает, что для приоритета памяти установлено значение MEMORY_PRIORITY_FOREGROUND, которое настраивается Win32k.sys при изменении фокуса), то к нему применяется повышение разноса приоритетов (PsPrioritySeparation) путем добавления его значения к верхушке нового приоритета. Это действие известно также как повышение приоритета потока первого плана, которое было рассмотрено ранее.

И наконец, ядро проверяет, не превышает ли этот новый вычисленный уровень приоритета текущий приоритет потока, и ограничивает это значение по верхнему значению в 15 единиц, чтобы избежать перехода в диапазон реального времени. Затем значение приоритета потока устанавливается в качестве нового текущего приоритета. Если применялось повышение приоритета на величину разноса приоритета потоков первого плана, это значение устанавливается в поле ForegroundBoost структуры данных KTHREAD, в результате чего значение PriorityDecrement становится равным значению разноса приоритетов.

Для повышений приоритета, связанных с событием AdjustBoost, ядро проверяет, не является ли текущий приоритет потока ниже значения AdjustIncrement (уровень приоритета, задаваемый при настройке потока) и не находится ли текущий приоритет потока ниже уровня в 13 единиц. Если так оно и есть, повышение приоритета потока отключено не будет, приоритет AdjustIncrement используется в качестве нового текущего приоритета, ограниченного максимумом в 13 единиц. Одновременно с этим значение повышения получает поле UnusualBoost структуры KTHREAD, в результате чего значение PriorityDecrement становится равным значению повышения приоритета, связанным с владением блокировкой.

Во всех случаях, где играет роль значение PriorityDecrement, квант времени потока также вычисляется заново, чтобы стать равным одному такту таймера, основанного на значении переменной KiLockQuantumTarget. Тем самым гарантируется, что повышение, превышающее обычное значение, будет утрачено после одного такта таймера, а не после обычных двух (или другого настроенного значения), как будет показано в следующем разделе. Точно так же все происходит и при запросе события AdjustBoost, но поток запускается с приоритетом 13 или 14 или с отключенным повышением.

Когда эта работа завершится, значение AdjustReason устанавливается в AdjustNone.

Удаление повышений

Удаление повышений осуществляется функцией KiDeferredReadyThread в виде простого вычисления заново применяемых (как показано в предыдущем разделе) повышений и квантов. Алгоритм начинается с проверки типа выполняемой настройки.

Для сценария AdjustNone это означает, что поток стал готов к выполнению, возможно, из-за вытеснения, квант времени потока будет пересчитан, если он уже достиг своей цели, но прерывания от таймера еще не замечено, поскольку поток был запущен с динамическим уровнем приоритета. Кроме того, будет пересчитан уровень приоритета потока. Для сценариев AdjustUnwait или AdjustBoost применительно к потоку, не относящемуся к потокам реального времени, ядро проверяет, не исчерпал ли поток свой квант до прерывания от таймера (точно так же, как при случае из предыдущего раздела). Если так оно и есть, или же если поток был запущен с базовым приоритетом равным 14 или выше, или же, если нет переменной PriorityDecrement и поток завершил ожидание, которое продолжалось дольше двух тактов таймера, квант потока пересчитывается, и то же самое происходит с его приоритетом.

Пересчет приоритета происходит в отношении потоков, не относящихся к потокам реального времени, и он осуществляется путем вычитания из текущего приоритета его повышения первого плана, вычитания повышения свыше обычного (сочетание этих двух элементов хранится в переменной PriorityDecrement) и, в завершение, вычитания единицы. И наконец, этот новый приоритет ограничивается базовым приоритетом в качестве низшей границы, и любой существующий декремент приоритета обнуляется (очищая повышения свыше обычного и повышения первого плана). Это означает, что в случае повышения, связанного с владением блокировкой, или любых рассмотренных повышений свыше обычного значения, все значение повышения утрачивается. С другой стороны, для обычного повышения AdjustUnwait приоритет естественным образом снижается на единицу за счет вычитания этой единицы. Это снижение, в конечном итоге, останавливается, как только путем проверки нижней границы будет достигнут базовый приоритет.

Есть еще один случай, когда повышение должно быть удалено с помощью функции KiRemoveBoostThread. Это особый случай удаления повышения, который происходит по правилу повышения, связанного с владением блокировкой, которое определяет, что настраиваемый поток должен утратить свое повышение, когда дарит свой текущий приоритет пробуждающемуся потоку (чтобы избежать сопровождения блокировки). Это удаление также используется для отмены повышения приоритета из-за целевых DPC-вызовов, а также для отмены повышения, связанного с блокировкой ресурсов исполняющей системы ERESOURCE. Единственной особенностью этой процедуры является то, что перед вычислением нового приоритета проявляется особое внимание разделению компонентов ForegroundBoost и UnusualBoost, составляющих значение PriorityDecrement с целью поддержки любых повышений на величину разноса GUI-потоков, связанных с их выходом на первый план, которые аккумулируются потоком. Такое поведение, ставшее новым для Windows 7, гарантирует то, что потоки, которые надеются на повышение, связанное с владением блокировкой, не будут вести себя неправильно при запуске на переднем плане или при возвращении на второй план.

На рис. 5.20 показан пример того, как обычные повышения удаляются из потока по истечении его кванта времени.

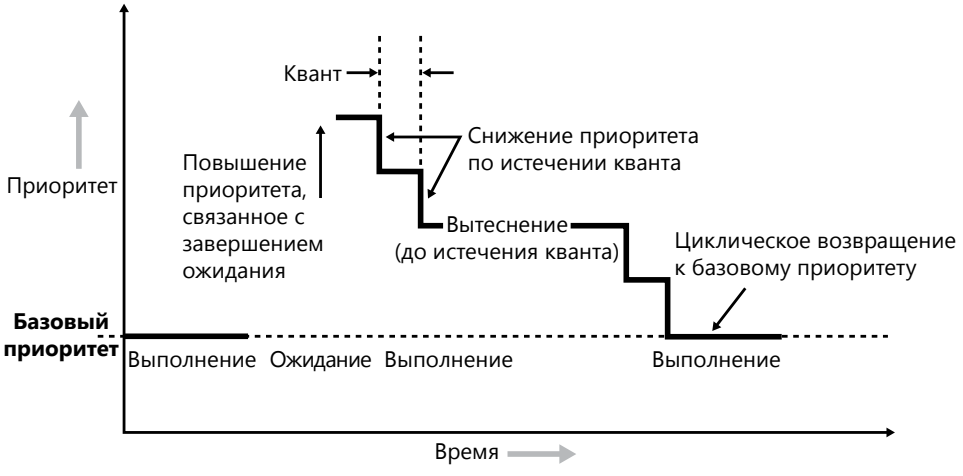


Рис. 5.20. Повышение и снижение приоритета

Повышения приоритетов для мультимедийных приложений и игр

Как было видно из последнего эксперимента, хотя повышения приоритета, связанного с перезагруженностью центрального процессора со стороны операционной системы Windows, может быть вполне достаточно для выхода потока из ненормально долгого состояния ожидания или из потенциальной взаимной блокировки, это повышение просто не может справиться с требованиями к ресурсам, предъявляемыми такими приложениями, как Windows Media Player или компьютерная игра в формате 3D.

Перескакивания и другие звуковые дефекты были в прошлом обычным источником раздражения со стороны пользователей Windows, а имеющийся в Windows аудиостек пользовательского режима только усугублял ситуацию, поскольку он еще больше повышал шансы на вытеснение потоков. Для борьбы с этим явлением в клиентские версии Windows была внедрена служба MMCSS, чьей целью было гарантировать проигрывание мультимедийного контента приложений, зарегистрированных с этой службой, без каких-либо сбоев.

MMCSS работает с вполне определенными задачами, включая следующие:

- аудио;
- захват;
- распределение;
- игры;
- проигрывание;
- аудио профессионального качества;
- задачи администратора многооконного режима.

ПРИМЕЧАНИЕ

Настройки MMCSS, включая список задач (который может быть изменен поставщиками для включения других специфических задач, соответствующей ресурсоемкости), можно найти в разделах реестра, входящих в раздел HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Multimedia\SystemProfile. Кроме того, параметр SystemResponsiveness позволяет осуществлять тонкую настройку того времени центрального процессора, которое служба MMCSS гарантирует потокам с низким уровнем приоритета.

В свою очередь, каждая из этих задач включает информацию о свойствах, отличающих их друг от друга. Одно из наиболее важных свойств для планирования потоков называется категорией планирования — Scheduling Category, которое является первичным фактором, определяющим приоритет потоков, зарегистрированных с MMCSS. В табл. 5.7 показаны различные категории планирования.

Таблица 5.7. Категории планирования

Категория	Приоритет	Описание
High (Высокая)	23–26	Потоки профессионального аудио (Pro Audio), запущенные с приоритетом выше, чем у других потоков на системе, за исключением критических системных потоков
Medium (Средняя)	16–22	Потоки, являющиеся частью приложений первого плана, например Windows Media Player
Low (Низкая)	8–15	Все остальные потоки, не являющиеся частью предыдущих категорий
Exhausted (Исчерпавших потоков)	1–7	Потоки, исчерпавшие свою долю времени центрального процессора, выполнение которых продолжится, только если не будут готовы к выполнению другие потоки с более высоким уровнем приоритета

Механизм, положенный в основу MMCSS, повышает приоритет потоков внутри зарегистрированного процесса до уровня, соответствующего их категории планирования и относительного приоритета внутри этой категории на гарантированный срок. Затем он снижает категорию этих потоков до Exhausted, чтобы другие, не относящиеся к мультимедийным приложениям потоки, также получили шанс на выполнение.

По умолчанию мультимедийные потоки получают 80 % доступного времени центрального процессора, а другие потоки получают 20 % этого времени (если взять в качестве примера 10 мс, это будет соответственно 8 мс и 2 мс). Сама служба MMCSS выполняется с приоритетом 27, поскольку ей нужно вытеснять любые Pro Audio-потоки с целью снижения их приоритета до категории Exhausted.

Следует иметь в виду, что по-прежнему повышает значения внутри KTHREAD (MMCSS осуществляет такой же системный вызов, как и любые другие приложения), а планировщик по-прежнему управляет этими потоками. Это просто их более высокие приоритеты, превращающие их выполнение на машине в почти непрерывное, поскольку они находятся в диапазоне реального времени и выше потоков, в которых запускается большинство пользовательских приложений.

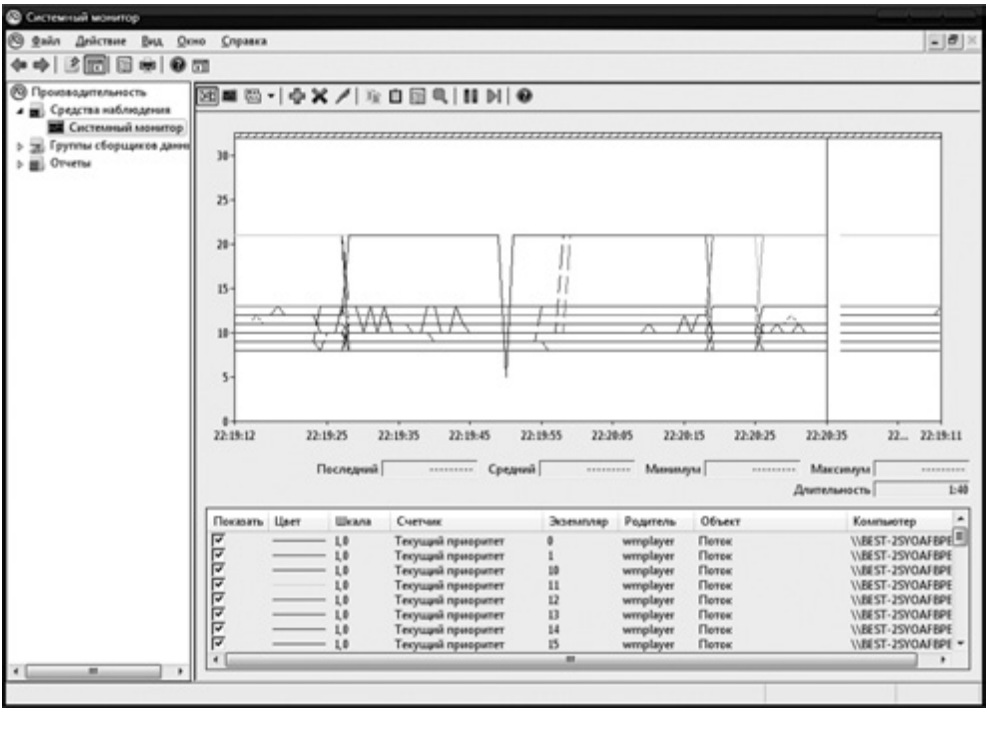
Как уже рассматривалось ранее, изменение относительных приоритетов потоков внутри процесса обычно не имеет смысла, и средств, позволяющих это сделать, не существует, поскольку только разработчики понимают важность различных потоков в их программах. С другой стороны, поскольку приложения должны самостоятельно регистрироваться со службой MMCSS и снабжать эту службу информацией о том, с какого рода потоком она имеет дело, у MMCSS есть необходимые данные для изменения этих относительных приоритетов потоков (и разработчики хорошо осведомлены о том, что это произойдет).

ЭКСПЕРИМЕНТ: «ПРОСЛУШИВАНИЕ» ПОВЫШЕНИЯ ПРИОРИТЕТА, СВЯЗАННОГО С РАБОТОЙ СЛУЖБЫ MMCSS

Теперь выполним такой же эксперимент, как и предыдущий, но без отключения службы MMCSS. Кроме того, посмотрим на оснастку Производительность (Performance), чтобы проверить приоритет потоков Windows Media Player.

1. Запустите Windows Media Player (другие программы проигрывания мультимедиа могут не пользоваться вызовами API-функций, которые требуют регистрации с MMCSS) и начните проигрывать какой-нибудь звуковой контент.
2. Если у вас мультипроцессорная машина, обязательно настройте родственность процесса Wmplayer.exe, чтобы он выполнялся только на одном центральном процессоре (поскольку будет использоваться только один рабочий поток CPUSTRES).
3. Запустите оснастку Производительность (Performance), выбрав в меню Пуск (Start) пункты Все программы (All Programs)▸ Администрирование (Administrative Tools)▸ Системный монитор (Performance Monitor). Щелкните на пунктах Средства наблюдения (Monitoring Tools)▸ Системный монитор (Performance Monitor).
4. Щелкните на кнопке панели инструментов Добавить (Add Counter) (или нажмите клавиши Ctrl+N), чтобы появилось диалоговое окно Добавить счетчики (Add Counters).
5. Выберите объект Поток (Thread), а затем выберите счетчик Текущий приоритет (Priority Current).
6. В поле со списком Экземпляры выбранного объекта (Instances) наберите Wmplayer, щелкните на кнопке Поиск (Search), а затем выберите все потоки этого приложения. Щелкните на кнопке Добавить (Add), а затем на кнопке OK.
7. Как и в предыдущем эксперименте, в меню Действие (Action) выберите пункт Свойства (Properties). На вкладке График (Graph) измените настройку Диапазон значений вертикальной шкалы Максимум: (Vertical Scale Maximum) на 31, а во вкладке Общие (General) в области Элементы диаграммы (Graph Elements) установите для поля Съем показаний каждые (Sample Every) интервал 1 и щелкните на кнопке OK. Вы должны увидеть внутри Wmplayer один или несколько потоков с приоритетом 21, которые будут постоянно выполняться, пока не будет высокоприоритетного потока, требующего времени центрального процессора после того, как они будут опущены до категории Exhausted.
8. Запустите утилиту Cpustres и установите уровень активности его потока Thread 1 на максимальный — Maximum.

9. Поднимите приоритет потока Thread 1 с обычного — Normal до критичного по времени — Time Critical.
10. Вы должны заметить существенное замедление работы системы, но проигрывание музыки продолжится. Время от времени вы сможете получить отклик от всей остальной системы. Воспользуйтесь этим для остановки утилиты Cpustres.
11. Если оснастке Производительность (Performance) не удалось захватить данные во время выполнения Cpustres, запустите ее еще раз, но воспользуйтесь наивысшим приоритетом — Highest, а не критичным по времени — Time Critical. Это изменение уменьшит замедление системы, но все еще потребует повышения приоритета от MMCSS. Поскольку при помещении мультимедийного потока в категорию Exhausted всегда будет поток с более высоким приоритетом, требующий времени центрального процессора (CPUSTRES), вы должны заметить, что приоритет 21 потока Wmplayer время от времени снижается, как показано на следующем рисунке.



Но функциональные возможности MMCSS на простом повышении приоритета не останавливаются. Из-за особенностей сетевых драйверов на Windows и NDIS-стека отложенные вызовы процедур — deferred procedure calls (DPC) — являются довольно распространенным механизмом для задержки работы после получения прерывания от сетевой карты. Так как DPC-вызовы запускаются на IRQL-уровне выше, чем у кода пользовательского режима (см. главу 3), долго работающий код драйвера сетевой карты по-прежнему может прервать проигрывание мультимедиа при сетевых передачах данных или, к примеру, при игре.

Поэтому MMCSS также отправляет специальную команду сетевому стеку, предписывая ему дросселировать сетевые пакеты, пока идет воспроизведение медиаконтета. Это дросселирование разработано для максимальной производительности воспроизведения ценой небольших потерь в пропускной способности сети (которая будет незаметна для сетевых операций, обычно осуществляемых во время проигрывания, например, при игре в онлайновую игру). Положенные в основу этой работы конкретные механизмы не имеют никакого отношения к планировщику, поэтому мы их рассматривать не будем.

ПРИМЕЧАНИЕ

Исходная реализация кода дросселирования сети имеет ряд конструкторских проблем, которые приводят к существенным потерям пропускной способности на машинах с сетевыми адаптерами в 1000 Мбит/с, особенно если на системе имеются несколько адаптеров (что часто бывает на материнский платах среднего ценового диапазона). Эти проблемы были проанализированы командами разработчиков MMCSS и сетей компании Microsoft и позже были устранены.

Переключения контекста

Контекст потока и процедура для переключения контекста сильно зависят от архитектуры процессора. Обычное переключение контекста требует сохранения и перезагрузки следующих данных:

- Указателя команд.
- Указателя стека ядра.
- Указателя на адресное пространство, в котором выполняется поток (каталог таблицы страниц процесса).

Ядро сохраняет эту информацию от старого потока, помещая ее в текущий (принадлежащий старому потоку) стек режима ядра, обновляя этот указатель стека и сохраняя указатель стека в структуре KTHREAD старого потока. Затем указатель стека ядра устанавливается на стек ядра нового потока, и загружается контекст нового потока. Если новый поток принадлежит другому процессу, он загружает свой адрес каталога таблица страниц в специальный регистр процессора, чтобы его адресное пространство стало доступным. Если APC-вызов ядра, который нуждается в доставке, задерживается, запрашивается прерывание уровня IRQL (см. главу 3). В противном случае управление передается восстановленному указателю команд нового потока, и новый поток возобновляет выполнение.

Сценарии планирования

В Windows ответ на вопрос «Кто получит центральный процессор?» основывается на приоритете потока, но как такой подход работает на практике? В следующих разделах будет показано, как именно на уровне потоков работает вытесняющая многозадачность, управляемая приоритетами.

Самостоятельное переключение

Сначала следует заметить, что поток может добровольно отказаться от использования процессора путем входа в состояние ожидания какого-нибудь объекта

(например, события, мьютекса, семафора и порта завершения ввода-вывода, процесса, потока, сообщения окна и т. п.) путем вызова одной из Windows-функций ожидания (WaitForSingleObject или WaitForMultipleObjects).

На рис. 5.21 показан поток, входящий в состояние ожидания, и выбор Windows нового потока для выполнения. На этом рисунке верхний блок (поток) самостоятельно уступает процессор, позволяя запуститься следующему потоку из очереди готовых потоков (что показано имеющимся над ним кольцом, полученным при переходе в колонку выполняемых потоков). Хотя при изучении данного рисунка может показаться, что приоритет уступившего процессор потока снизился, на самом деле это не так, поток просто был перемещен в очередь ожидания того объекта, которого он ожидает.

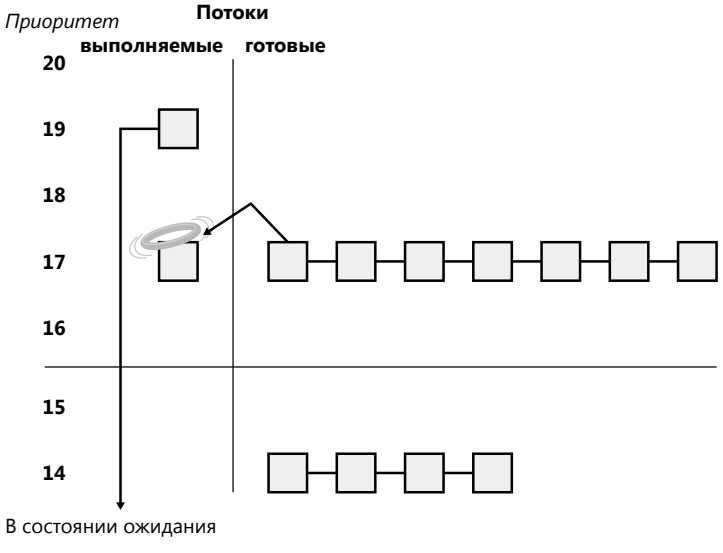


Рис. 5.21. Самостоятельное переключение

Вытеснение

В этом сценарии планировки поток с более низким приоритетом вытесняется, когда становится готовым к выполнению поток с более высоким приоритетом. Такая ситуация может сложиться по двум причинам:

- Поток с более высоким приоритетом завершил ожидание. (Произошло ожидаемое им событие.)
- Произошло повышение или снижение приоритета потока.

В любом из этих случаев Windows должна определить, должен ли текущий выполняемый поток продолжить свое выполнение, или он должен быть вытеснен, позволив выполняться потоку с более высоким приоритетом.

Когда поток вытесняется, он помещается в начало очереди готовых потоков для того уровня приоритета, с которым он выполнялся. Эта ситуация показана на рис. 5.22.

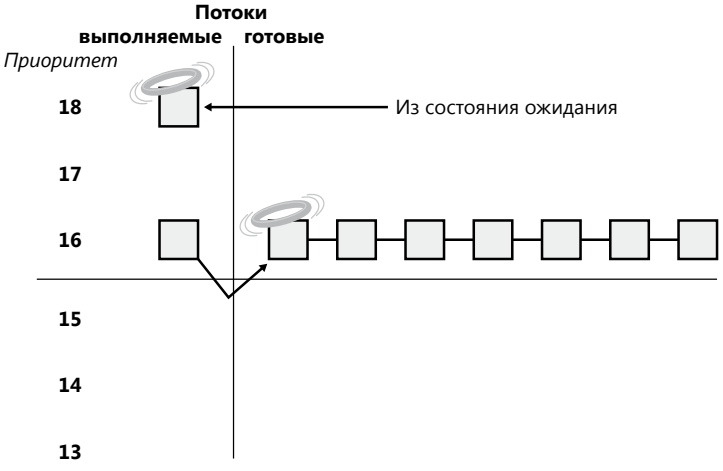


Рис. 5.22. Планирование с вытеснением потоков

ПРИМЕЧАНИЕ

Потоки, выполняемые в пользовательском режиме, могут вытеснять потоки, выполняемые в режиме ядра — режим, в котором выполняются потоки, значения не имеет. Определяющим фактором является приоритет потока.

На рис. 5.22 поток с уровнем приоритета 18 выходит из состояния ожидания и снова получает в свое распоряжение центральный процессор, выталкивая выполняемый поток (с уровнем приоритета 16) в начало очереди готовых потоков. Следует заметить, что вытолкнутый поток помещается не в конец, а в начало очереди; когда вытеснивший его поток завершит свое выполнение, вытолкнутый поток может завершить свой квант выполнения.

Истечение кванта времени

Когда у выполняемого потока истекает квант времени, Windows должна определить, нужно ли снижать приоритет потока, а затем определить, должно ли быть спланировано выполнение на процессоре другого потока.

Если приоритет потока снижается, Windows ищет для планирования выполнения самый подходящий поток. (Например, наиболее подходящим может быть поток в очереди готовых потоков, имеющих более высокий уровень приоритета, чем новый приоритет для текущего выполняемого потока.) Если приоритет потока не снижается и в очереди готовых потоков имеются потоки с таким же уровнем приоритета, Windows выбирает следующий поток из очереди готовых потоков с таким же уровнем приоритета и перемещает ранее выполнявшийся поток в конец очереди (задавая ему новое значение кванта и изменяя его состояние с выполняющегося на готовое). Этот случай показан на рис. 5.23. Если другого готового к выполнению потока с таким же уровнем приоритета не имеется, поток получает для выполнения еще один квант времени.

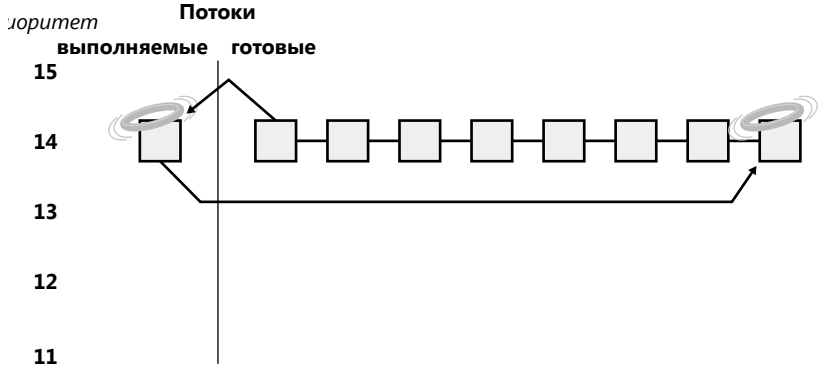


Рис. 5.23. Планирование потоков при истечении кванта времени

Как уже было показано, вместо того чтобы просто полагаться при планировании выполнения потоков на квант времени, основанный на работе интервального таймера, Windows использует для обслуживания квантовых целей точный счетчик тактовых циклов центрального процессора. Нам следует рассмотреть еще один неупомянутый фактор, который Windows также использует в своих расчетах, применимо ли в данном случае истечение кванта времени к потоку.

Использование модели планирования, которая полагается только на интервальный таймер, может привести к следующим ситуациям:

- Потоки А и Б стали готовы к выполнению в середине интервала. (Код планирования запускается не по каждому интервалу времени, поэтому такое часто случается.)
- Поток А начал выполняться, но его выполнение на какое-то время было прервано. Время, затраченное на обработку прерывания, потоку не возмещается.
- Обработка прерывания завершается, и поток А снова запускается на выполнение, но быстро достигает следующего интервала таймера. Планировщик может только предположить, что поток А выполнялся все это время, и теперь переключается на выполнение потока Б.
- Поток Б начинает выполняться и имеет шанс на выполнение в течение полного интервала таймера (если не брать в расчет вытеснение или обработку прерывания).

При таком сценарии развития событий поток А был незаслуженно наказан двумя различными способами. Сначала время, затраченное на обработку прерывания от устройства, было зачтено как его собственное время использования центрального процессора, даже при том, что поток, возможно, из-за прерывания так ничего и не сделал. (Следует напомнить, что прерывания обрабатываются в контексте того потока, который выполнялся на данный момент.) Он также был незаслуженно наказан на то время, пока система простаивала внутри интервала таймера, до того, как он был снова спланирован на выполнение.

Именно такое развитие событий показано на рис. 5.24.

Поскольку Windows ведет точный подсчет количества тактовых циклов центрального процессора, затраченных на выполнение той работы, которая была спланирована для потока (без учета прерываний), и поскольку она хранит

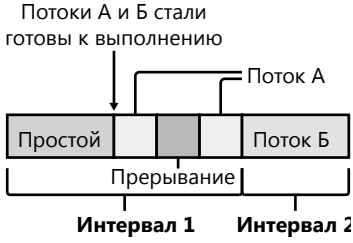


Рис. 5.24. Несправедливое квантование времени в предыдущих версиях Windows

квантовую цель в тактовых циклах, которые должны быть потрачены потоком до завершения его кванта времени, оба несправедливых решения, которые могли бы быть приняты в отношении потока А в Windows, не принимаются.

Вместо этого происходит следующее:

- Потоки А и Б стали готовы к выполнению в середине интервала.
- Поток А начал выполняться, но его выполнение на какое-то время было прервано. Тактовые циклы центрального процессора, затраченные на обработку прерывания, потоку не возмещаются.
- Обработка прерывания завершается, и поток А снова запускается на выполнение, но быстро достигает следующего интервала таймера. Планировщик смотрит на количество тактовых циклов центрального процессора, выделенных потоку, и сравнивает его с ожидаемым количеством тактовых циклов центрального процессора, которые должны были быть выделены до конца кванта времени.
- Поскольку предшествующее количество намного меньше того, каким оно должно быть, планировщик предполагает, что поток А начал выполняться в середине интервала таймера и, кроме того, его выполнение могло быть прервано.
- Поток А получает повышение кванта своего времени на еще один интервал таймера, и квантовая цель пересчитывается. Теперь у потока А есть шанс выполняться в течение полного интервала таймера.
- В следующий интервал таймера поток А выберет свой квант времени, и теперь шанс на выполнение получит поток Б.

Именно такое развитие событий показано на рис. 5.25.

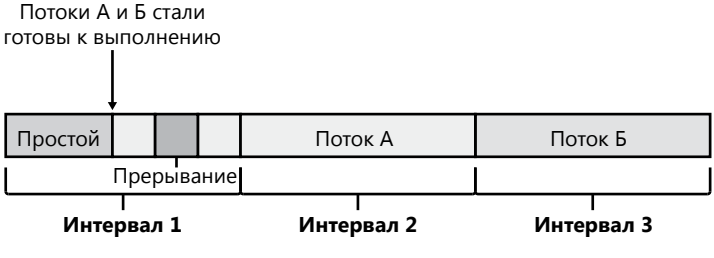


Рис. 5.25. Справедливое квантование времени в текущих версиях Windows

Завершение выполнения потока

Когда поток завершает свое выполнение (либо по причине возвращения из своей основной процедуры, называемой ExitThread, либо по причине его уничтожения с помощью функции TerminateThread), он переходит из состояния выполнения в состояние завершения. Если у объекта потока нет открытых дескрипторов, поток удаляется из списка потоков процесса, а связанные с ним структуры данных освобождают выделенные им ресурсы.

Потоки простоя

Когда у центрального процессора нет потоков, готовых к выполнению, Windows запускает на этом центральном процессоре поток простоя. У каждого центрального процессора есть свой собственный, выделенный поток простоя, поскольку на мультипроцессорных системах один центральный процессор может выполнять поток, в то время как у другого центрального процессор может не быть потоков для выполнения. Каждый поток простоя центрального процессора находится через указатель в PRCB-блоке этого процессора.

Все потоки простоя принадлежат процессу простоя. Во многих отношениях процесс простоя и потоки простоя являются особым случаем. Они, конечно же, представлены структурами EPROCESS/KPROCESS и ETHREAD/KTHREAD, но не являются процессами диспетчера исполняющей системы и объектами потоков. Процесс простоя также не входит в список системных процессов. (Именно поэтому он не появляется в выводе команды отладчика ядра !process 0 0.) Но поток или потоки простоя и их процесс можно обнаружить другими способами.

ЭКСПЕРИМЕНТ: ВЫВОД СТРУКТУРЫ ПОТОКОВ ПРОСТОЯ И ИХ ПРОЦЕССА

Структуры потока и процесса простоя могут быть найдены в отладчике ядра с помощью команды !pcr. «PCR» является аббревиатурой «processor control region» — области управления процессором. Эта команда выводит поднабор информации из PCR, а также из связанного с этой областью блока управления процессора — processor control block (PRCB). Команда !pcr получает один числовой аргумент, представляющий собой номер центрального процессора, чья область PCR будет выведена.

Загрузочным процессором является процессор с номером 0, и он всегда присутствует в системе, поэтому команда !pcr 0 будет работать всегда. В следующем выводе показан результат запуска этой команды в виде дампа памяти, взятого из 64-разрядной, четырехпроцессорной системы:

```
3: kd> !pcr 0
KPCR for Processor 0 at fffff800039fdd00:
    Major 1 Minor 1
        NtTib.ExceptionList: fffff80000b95000
            NtTib.StackBase: fffff80000b96080
           NtTib.StackLimit: 000000000008e2d8
         NtTib.SubSystemTib: fffff800039fdd00
              NtTib.Version: 00000000039fde80
```

```
NtTib.UserPointer: fffff800039fe4f0
    NtTib.SelfTib: 000000007efdb000
          SelfPcr: 0000000000000000
             Prcb: fffff800039fde80
             Irql: 0000000000000000
              IRR: 0000000000000000
              IDR: 0000000000000000
    InterruptMode: 0000000000000000
              IDT: 0000000000000000
              GDT: 0000000000000000
              TSS: 0000000000000000
    CurrentThread: fffffa8007aa8060
       NextThread: 0000000000000000
       IdleThread: fffff80003a0bcc0
        DpcQueue:
```

Из этого вывода видно, что на время получения дампа памяти CPU 0 выполнял процесс, отличный от своего процесса простоя, поскольку указатели CurrentThread и IdleThread имеют разные значения. (Если у вас мультипроцессорная система, можно попробовать запустить команды !pcr 1, !pcr 2 и т. д., до тех пор, пока не закончатся процессоры, наблюдая за тем, что у всех указатели IdleThread имеют разные значения.)

Теперь воспользуйтесь командой !thread в отношении показанного адреса потока простоя:

```
3: kd> !thread fffff80003a0bcc0
THREAD fffff80003a0bcc0 Cid 0000.0000 Teb: 0000000000000000 Win32Thread:
0000000000000000
  RUNNING on processor 0
Not impersonating
DeviceMap                 fffff8a000008aa0
Owning Process            fffff80003a0c1c0       Image:         Idle
Attached Process          fffffa800792a040       Image:         System
Wait Start TickCount      50774016       Ticks: 12213 (0:00:03:10.828)
Context Switch Count      1147613282
UserTime                  00:00:00.000
KernelTime                8 Days 07:21:56.656
Win32 Start Address nt!KiIdleLoop (0xfffff8000387f910)
Stack Init fffff80000b9cdb0 Current fffff80000b9cd40
Base fffff80000b9d000 Limit fffff80000b97000 Call 0
Priority 16 BasePriority 0 UnusualBoost 0 ForegroundBoost 0 IoPriority 0 PagePriority 0
Child-SP          RetAddr           : Args to Child     [...]: Call Site
fffff800'00b9cd80 00000000'00000000 : fffff800'00b9d000 [...]: nt!KiIdleLoop+0x10d
```

И наконец, воспользуйтесь командой !process в отношении адреса, указанного для процесса-владельца — «Owning Process», показанного в предыдущем выводе. Для краткости мы добавим еще один аргумент со значением 3, который заставит команду !process вывести только минимум информации для каждого потока:

```
3: kd> !process fffff80003a0c1c0 3
PROCESS fffff80003a0c1c0
    SessionId: none  Cid: 0000    Peb: 00000000  ParentCid: 0000
    DirBase: 00187000  ObjectTable: fffff8a000001630  HandleCount: 1338.
    Image: Idle
    VadRoot fffffa8007846c00 Vads 1 Clone 0 Private 1. Modified 0. Locked 0.
    DeviceMap 0000000000000000
    Token                             fffff8a000004a40
    ElapsedTime                       00:00:00.000
    UserTime                          00:00:00.000
    KernelTime                        00:00:00.000
    QuotaPoolUsage[PagedPool]         0
    QuotaPoolUsage[NonPagedPool]      0
    Working Set Sizes (now,min,max)  (6, 50, 450) (24KB, 200KB, 1800KB)
    PeakWorkingSetSize                6
    VirtualSize                       0 Mb
    PeakVirtualSize                   0 Mb
    PageFaultCount                    1
    MemoryPriority                    BACKGROUND
    BasePriority                      0
    CommitCharge                      0
THREAD fffff80003a0bcc0  Cid 0000.0000  Teb: 0000000000000000 Win32Thread:
    0000000000000000
  RUNNING on processor 0
THREAD fffff8800310afc0  Cid 0000.0000  Teb: 0000000000000000 Win32Thread:
    0000000000000000
  RUNNING on processor 1
THREAD fffff8800317bfc0  Cid 0000.0000  Teb: 0000000000000000 Win32Thread:
    0000000000000000
  RUNNING on processor 2
THREAD fffff880031ecfc0  Cid 0000.0000  Teb: 0000000000000000 Win32Thread:
    0000000000000000
  RUNNING on processor 3
```

Эти адреса процесса и потока могут быть использованы с командами dt nt!_EPROCESS, dt nt!_KTHREAD, а также с другими аналогичными командами. ■

Предыдущий эксперимент показал ряд аномалий, связанных с процессом простоя и его потоками. Отладчик показывает для «Image» (образа) имя «Idle» (простой) (которое взялось из компонента ImageFileName структуры EPROCESS).Но различные утилиты Windows в своих отчетах показывают, что процесс простоя использует разные имена. Диспетчер задач (Task Manager) и Process Explorer называют его «System Idle Process», а Tlist называет его «System Process». Идентификатор процесса (process ID) и идентификаторы потоков (thread ID) (которые в выводе отладчика фигурируют как клиентские идентификаторы «client Ids», или «Cid») равны нулю, так же как PEB- и TEB-указатели, и есть еще множество других полей в процессе простоя или в его потоках, в значениях которых может быть показан 0. Это происходит потому, что процесс простоя не имеет адресного пространства режима пользователя и его потоки не выполняют

кода в режиме пользователя, поэтому им и не нужны различные данные, необходимые для управления средой пользовательского режима. Кроме того, процесс простоя не является объектом процесса диспетчера объектов. Вместо этого исходные структуры потока простоя и процесса простоя размещены статически и используются для самозагрузки системы перед тем, как будут инициализированы диспетчер процессов и диспетчер объектов. Последующие структуры потока простоя размещаются динамически (в виде простых размещений из невыгружаемого пула, в обход диспетчера объектов), как только в дело вступают дополнительные процессоры. После инициализации управления процессами для ссылки на процесс простоя используется специальная переменная PsIdleProcess.

Наверное, самой интересной аномалией, относящейся к процессу простоя, является то, что Windows сообщает, что у процесса простоя уровень приоритета равен 0 (как показано ранее, на x64 системах это значение равно 16). Но в действительности у приоритетов потоков простоя нет значения, потому что такие потоки выбираются для диспетчеризации, только если нет никаких других потоков, готовых к выполнению. Их приоритет никогда не сравнивается с приоритетами других потоков, не используется для помещения потока простоя в очередь готовых потоков, потоки простоя никогда не стоят ни в каких очередях готовых потоков. (Запомните, только один поток в каждой системе Windows действительно выполняется с приоритетом 0 — это поток обнуления страниц (zero page thread).)

Потоки простоя — не только особый случай в понятиях выбора на выполнение, они также являются особыми случаями для вытеснения. Процедура потоков простоя KiIdleLoop выполняет ряд операций, которые препятствуют его вытеснению другими потоками в обычном порядке. Когда на процессоре нет готовых к выполнению потоков, не относящихся к потокам простоя, этот процессор помечается в PRCB как простаивающий. После этого, если поток выбран для выполнения на простаивающем процессоре, адрес потока хранится в указателе следующего потока NextThread в PRCB простаивающего процессора. Поток простоя проверяет этот указатель при каждом проходе своего цикла.

Хотя кое-какие детали в последовательности варьируются от архитектуры к архитектуре, основная последовательность операций потока простоя представляет собой следующую картину:

1. Разрешение прерываний на короткий промежуток времени, позволяющее доставить любые отложенные прерывания, а затем новое их запрещение (с использованием инструкций STI и CLI —на процессорах x86 и x64). Наличие этой операции обусловлено тем, что значительная часть выполнения потока простоя происходит с отключенными прерываниями.
2. В отладочных версиях некоторых архитектур проверка, не пытается ли отладчик ядра вклиниться в систему, и если так оно и есть, предоставление ему соответствующего доступа.
3. Проверка наличия на процессоре отложенных вызовов процедур DPC (рассмотренных в главе 3). DPC-вызовы могут ожидать решения, если не было сгенерировано DPC-прерывание, когда они были поставлены в очередь. При наличии DPC-вызовов, ожидающих решения, цикл простоя вызывает для их доставки функцию KiRetireDpcList. Это же действие будет выполняться

по истечении интервала таймера, так же как обработка отложенных готовых потоков; последнее действие рассматривается в следующем далее разделе, посвященном планированию на мультипроцессорных системах. Вход в функцию KiRetireDpcList должен быть осуществлен при отключенных прерываниях, именно поэтому прерывания остаются отключенными после выполнения действия 1. При выходе из функции KiRetireDpcList прерывания также остаются отключенными.

4. Проверка, был ли для процессора выбран следующий поток для выполнения и если такой поток есть, диспетчеризация этого потока. Это может быть в том случае, если, к примеру, обрабатываемый в действии 3 DPC-вызов или истечение времени таймера удовлетворили условия ожидающего потока или если другой процессор выбрал поток на выполнение для этого процессора, который в этот момент уже отрабатывал цикл простоя.
5. Проверка, если таковая требуется, наличия потоков, готовых к выполнению на других процессорах, и если это возможно, локальное планирование работы одного из них. (Эта операция рассматривается в следующем далее разделе «Планировщик простоя».)
6. Вызов зарегистрированной процедуры управления электропитанием простаивающего процессора (в случае необходимости выполнения любых функций управления электропитанием), которая является либо драйвером электропитания процессора (таким как intelppm.sys), либо находится в HAL, если такой драйвер недоступен.

Выбор потока

Когда логическому процессору нужно выбрать следующий выполняемый процесс, он вызывает функцию планировщика KiSelectNextThread. Это может происходить по различным сценариям:

- Произошло изменение жестко заданной родственности, что сделало текущий выполняемый поток или поток, находящийся в состоянии повышенной готовности, неподходящим для выполнения на его выбранном логическом процессоре, поэтому должен быть выбран другой процессор.
- Текущий выполняемый поток исчерпал свой квант, и SMT-набор, на котором он в данный момент выполнялся, теперь занят, в то время как другие SMT-наборы в составе идеального узла полностью простаивают. Планировщик осуществляет миграцию текущего потока, связанную с истечением его кванта, поэтому должен быть выбран другой поток.
- Операция ожидания завершена, и в регистре статуса ожидания находились незавершенные операции планирования (иными словами, были установлены биты приоритета — Priority и (или) родственности — Affinity).

При таких сценариях планировщик ведет себя следующим образом:

- Вызывает функцию KiSelectReadyThread, чтобы найти следующий готовый поток, который должен быть выполнен процессором, и проверяет, был ли такой поток найден.

- Если готовый поток не был найден, включается планировщик простоя, и для выполнения выбирается поток простоя.
- Или, если готовый поток был найден, он помещается в состояние Standby и устанавливается в качестве следующего потока — NextThread в блоке KPRCB логического процессора.

Операция выбора следующего потока KiSelectNextThread выполняется только тогда, когда логический процессор нуждается в подборе, но еще не в выполнении следующего планируемого потока (по причине чего поток войдет в состояние Standby). Но в другой раз логический процессор заинтересован в немедленном запуске следующего готового потока или выполнить другое действие, если он не доступен (вместо того, чтобы простаивать), как при возникновении следующих обстоятельств:

- Произошло изменение приоритета, и теперь текущий поток, находящийся в состоянии повышенной готовности, или выполняющийся поток не является готовым потоком, имеющим самый высокий приоритет на выбранном им логическом процессоре, и теперь должен быть выполнен поток, имеющий более высокий приоритет.
- Поток явным образом уступил свою очередь с помощью функции YieldProcessor или NtYieldExecution, и в готовности к выполнению мог бы быть другой поток.
- Квант времени текущего потока истек, и другие потоки с тем же уровнем приоритета нуждаются в получении своего шанса на выполнение.
- Поток утратил свое повышение приоритета, вызывая тем самым аналогичное изменение приоритетов в уже рассмотренном сценарии.
- Запущен планировщик простоя, и нужно проверить, не появился ли в интервале между запросом на планирование простоя и запуском планировщика простоя готовый к выполнению поток.

Простой способ запомнить разницу между тем, какая из процедур запускается, заключается в проверке, *должен* ли логический процессор выполнить другой поток (в таком случае вызывается процедура KiSelectNextThread) или *может ли* он, если это возможно, выполнить другой поток (в таком случае вызывается процедура KiSelectReadyThread).

Каждый процессор имеет свою собственную базу данных потоков, готовых к выполнению (принадлежащие базе данных диспетчера находятся в KPRCB). Процедура KiSelectReadyThread может просто проверить очереди текущего логического процессора, забрав первый найденный поток с самым высоким приоритетом, если этот приоритет не ниже, чем у текущего выполняемого потока (в зависимости от того, разрешено ли выполнение текущего потока, что может не относиться к случаю развития событий по сценарию выбора следующего потока — KiSelectNextThread). Если потока с более высоким приоритетом нет (или вообще нет готовых к выполнению потоков), то возвращенного потока не будет.

Планировщик простоя

Как только запускается поток простоя, он проверяет, включен ли планировщик простоя, как в одном из сценариев, рассмотренном в предыдущем разделе.

Если планировщик простоя включен, поток простоя путем вызова процедуры KiSearchForNewThread начинает сканировать очереди готовых потоков других процессоров в поисках потоков, которые он может запустить. Следует учесть, что расходы на выполнение, связанные с этой операцией, не засчитываются как время потока простоя, а учитываются как время прерывания и DPC-вызова (относятся на счет процессора), поэтому время планирования простоя считается системным временем.

Алгоритм процедуры KiSearchForNewThread, который основан на функциях, рассмотренных ранее в разделе «Выбор потока», будет рассмотрен в следующем далее разделе.

Мультипроцессорные системы

В однопроцессорной системе планирование осуществляется сравнительно просто: всегда запускается поток с самым высоким приоритетом, требующий выполнения. В мультипроцессорной системе планирование осуществляется гораздо сложнее, поскольку Windows предпринимает попытку спланировать выполнение потоков на наиболее оптимальном для потока процессоре, принимая во внимание предпочитаемые потоком процессоры и те процессоры, на которых он уже выполнялся, а также конфигурацию мультипроцессорной системы. Поэтому, хотя Windows пытается спланировать работу готовых к выполнению потоков, имеющих самый высокий приоритет на всех доступных центральных процессорах, она гарантирует только то, что где-нибудь ею будет запущен один из потоков, имеющих самый высокий приоритет.

Перед тем как рассматривать конкретные алгоритмы, используемые для выбора, какие потоки когда и где запускать, давайте изучим дополнительную информацию, поддерживаемую системой Windows для отслеживания состояния потока и процессора на мультипроцессорных системах и три различных типа мультипроцессорных систем, поддерживаемых Windows (SMT, многоядерные и NUMA).

Наборы пакетов и SMT-наборы

Для принятия правильных решений по планированию потоков при работе с топологиями логических процессоров в Windows в блоке KPRCB используется пять полей. Первое поле, CoresPerPhysicalProcessor, определяет, является ли этот логический процессор частью многоядерного пакета, и его значение вычисляется из CPUID, возвращенного процессором, и округляется до степени числа два. Второе поле, LogicalProcessorsPerCore, определяет, является ли логический процессор частью SMT-набора, например, на процессоре Intel с включенной технологией HyperThreading, и его значение также запрашивается через CPUID и округляется. Перемножение этих двух чисел дает количество логических процессоров в пакете или в реальном физическом процессоре, вставленном в гнездо. Располагая этим количеством, каждый блок PRCB может затем заполнить свое значение PackageProcessorSet, которое является маской родственности, дающей описание, какие еще логические процессоры внутри этой группы (поскольку

пакеты ограничиваются группами) принадлежат одному и тому же физическому процессору. По аналогии с этим, значение CoreProcessorSet объединяет другие логические процессоры одного и того же ядра, что также называется SMT-набором. И наконец, значение GroupSetMember определяет, какая битовая маска внутри текущей группы процессоров идентифицирует тот или иной логический процессор. Например, логический процессор 3 обычно имеет значение GroupSetMember, равное 8 (2 в степени 3).

ЭКСПЕРИМЕНТ: ПРОСМОТР ИНФОРМАЦИИ О ЛОГИЧЕСКОМ ПРОЦЕССОРЕ

Изучить информацию, предоставляемую Windows для SMT-процессоров, можно с помощью команды отладчика ядра !smt. Следующий вывод получен из двухъядерной системы Intel Core i5 с SMT (с четырьмя логическими процессорами):

```
SMT Summary:
KeActiveProcessors:
****------------------------------------------------------ (000000000000000f)
KiIdleSummary:
-*-*------------------------------------------------------ (000000000000000a)
-------------------------------------------------------- (0000000000000000)
-------------------------------------------------------- (0000000000000000)
-------------------------------------------------------- (0000000000000000)
No PRCB             SMT Set                              APIC Id
  0 fffff8000324ae80 **------------------------(0000000000000003) 0x00000000
  1 fffff880009e5180 **------------------------(0000000000000003) 0x00000001
  2 fffff88002f65180 --**----------------------(000000000000000c) 0x00000002
  3 fffff88002fd7180 --**----------------------(000000000000000c) 0x00000003
Maximum cores per physical processor:   8
Maximum logical processors per core:    2
```

NUMA-системы

Windows поддерживает еще один тип мультипроцессорных систем, архитектуру с технологией доступа к неоднородной памяти — nonuniform memory access (NUMA). В NUMA-системе процессоры группируются вместе в небольшие модули, которые называются узлами. У каждого узла есть свои собственные процессоры и память, и он подключен к более крупной системе через объединительную шину с когерентной кэш-памятью. Эти системы называются «неоднородными», поскольку у каждого узла имеется своя собственная локальная высокоскоростная память. Хотя любой процессор в любом узле может получить доступ ко всей памяти, доступ к локальной память узла осуществляется намного быстрее.

Ядро предоставляет информацию о каждом узле в NUMA-системе в структуре данных, называемой KNODE. Переменная ядра KeNodeBlock является массивом указателей на KNODE-структуры для каждого узла. Формат KNODE-структуры может быть показан в отладчике ядра с помощью команды dt, как показано в следующем примере:

```
lkd> dt nt!_KNODE
   +0x000 PagedPoolSListHead : _SLIST_HEADER
   +0x008 NonPagedPoolSListHead : [3] _SLIST_HEADER
   +0x020 Affinity         : _GROUP_AFFINITY
   +0x02c ProximityId      : Uint4B
   +0x030 NodeNumber       : Uint2B
...
   +0x060 ParkLock         : Int4B
   +0x064 NodePad1         : Uint4B
```

ЭКСПЕРИМЕНТ: ПРОСМОТР NUMA-ИНФОРМАЦИИ

Информацию, предоставляемую Windows для каждого узла в NUMA-системе, можно просмотреть с помощью команды отладчика ядра !numa. Следующий частичный вывод получен из 64-процессорной NUMA-системы от Hewlett-Packard с четырьмя процессорами на узел:

```
26: kd> !numa
NUMA Summary:
------------
Number of NUMA nodes : 16
Number of Processors : 64
MmAvailablePages     : 0x03F55E67
KeActiveProcessors   : ************************************************************
                       (ffffffffffffffff)

NODE 0 (E000000084261900):
    ProcessorMask    : ****------------------------------------------------
...
NODE 1 (E0000145FF992200):
    ProcessorMask    : ----****------------------------------------------------
...
```

■

Приложения, которым требуется получить от NUMA-систем наивысшую производительность, могут установить маску родственности для ограничения процесса процессорами определенного узла, хотя Windows уже ограничивает чуть ли не все потоки единственным NUMA-узлом из-за своих алгоритмов планирования, осведомленных о NUMA-системе (см. раздел «Выбор процессора»).

Назначение группы процессоров

При запросе топологии системы для построения разнообразных взаимоотношений между логическими процессорами, SMT-наборами, многоядерными пакетами и физическими сокетами Windows назначает процессоры соответствующей группе, которая опишет их родственность (посредством рассмотренной ранее расширенной маски родственности).

Эта работа делается процедурой KePerformGroupConfiguration, которая вызывается при инициализации до выполнения любой другой работы на этапе 1. Следует заметить, что независимо от следующих далее этапов назначения группы, NUMA-узел 0 всегда, несмотря ни на что, назначается группе 0.

Сначала функция запрашивает все найденные узлы (KeNumberNodes) и вычисляет емкость каждого узла (то есть сколько логических процессоров может быть частью узла). Это значение сохраняется в переменной MaximumProcessors блока KeNodeBlock, который идентифицирует все NUMA-узлы системы. Если система поддерживает идентификаторы близости NUMA Proximity ID, для каждого узла запрашивается также идентификатор близости (proximity ID), который сохраняется в блоке узла. Затем назначается массив расстояний NUMA (NUMA distance array, KeNodeDistance) и, как было рассмотрено в главе 3, вычисляется расстояние между всеми NUMA-узлами.

Следующая череда шагов связана с особенными настройками пользовательской конфигурации, которые заменяют NUMA-назначения, определяемые по умолчанию. Например, на системах с установленной технологией Hyper-V (с гипервизором, настроенным на автозапуск) будет доступна только одна группа процессоров, и все NUMA-узлы (которые могут поместиться) будут связаны с группой 0. Это означает, что сценарии Hyper-V не могут в данный момент воспользоваться машинами с более чем 64 процессорами.

Затем функция проверяет наличие любых данных о назначении статических групп, переданных загрузчиком (и поэтому настраиваются пользователем). Эти данные определяют информацию о близости и групповые назначения для каждого NUMA-узла.

ПРИМЕЧАНИЕ

Пользователи, работающие с крупными NUMA-серверами, которые могут нуждаться в пользовательском управлении информацией о близости и групповых назначениях с целью проведения тестирования и проверок, могут ввести эти данные через параметры реестра Group Assignment и Node Distance в разделе HKLM\SYSTEM\CurrentControlSet\Control\NUMA. Точный формат этих данных включает итоговую сумму, за которой следует массив идентификаторов близости (proximity ID) и групповых назначений, которые являются 32-разрядными значениями.

Перед тем как считать эти данные достоверными, ядро запрашивает идентификатор близости, чтобы сопоставить с номером узла, а затем, как это требуется, связывает номера групп. Затем оно убеждается в том, что NUMA-узел 0 связан с группой 0, и в том, что емкость всех NUMA-узлов согласуется с размером группы. И наконец, функция проверяет, у скольких групп все еще имеется остаточная емкость.

Затем ядро в динамическом режиме пытается назначить NUMA-узлы группам, признавая любые статически настроенные узлы, если они были переданы так, как это описывалось ранее. Обычно ядро пытается минимизировать количество создаваемых групп, объединяя в каждой группе как можно больше NUMA-узлов. Но если такое поведение нежелательно, оно может быть скорректировано с помощью параметра загрузчика /MAXGROUP, который настраивается через BCD-элемент maxgroup. Включение этого параметра переопределяет поведение по умолчанию и заставляет алгоритм распространять как можно больше NUMA-узлов в как можно большем количестве групп (с учетом того, что в настоящее время реализовано ограничение в количестве групп, равное четырем). Если имеется только один узел или если все узлы могут поместиться в одной группе

(и элемент maxgroup отключен), система осуществляет настройки по умолчанию и назначает все узлы группе 0.

Если имеется более одного узла, Windows проверяет статические дистанции NUMA-узлов (если таковые имеются), а затем сортирует все узлы по их емкости, чтобы первым стал самый крупный узел. А в режиме минимизации групп ядро путем сложения всех емкостей определяет, какое максимальное количество процессоров может быть в системе. Путем деления результата на количество процессоров в каждой группе ядро предполагает, что на машине будет именно такое общее количество групп (ограниченное максимумом в четыре группы). В режиме максимизации групп исходной оценкой будет то, что групп будет столько же, сколько и узлов (опять же ограничивая количество групп числом 4).

Теперь ядро приступает к завершающему процессу назначений. Утверждаются все ранее сделанные фиксированные назначения, и для них создаются группы. Затем все NUMA-узлы перегруппировываются для минимизации дистанции между различными узлами в группе. Иными словами, более близкие узлы помещаются в одну группу и сортируются по расстояниям. После этого для любого динамически сконфигурированного узла осуществляются одни и те же процессы для групповых назначений. И наконец, любые оставшиеся пустые узлы назначаются группе 0.

Логические процессоры, приходящиеся на каждую группу

Обычно, как уже ранее было рассмотрено, Windows назначает 64 процессора на каждую группу, но эта конфигурация также может быть подстроена под использование различных вариантов нагрузки, например, с помощью параметра /GROUPSIZE, который настраивается через BCD-элемент groupsize. Указывая число, являющееся степенью двойки, можно заставить группу содержать количество процессоров меньшее обычного, для таких целей, как тестирование осведомленности о группах в системе (система с 8 логическими процессорами может быть представлена имеющей 1, 2 или 4 группы). Чтобы принудительно решить данный вопрос, параметр /FORCEGROUPAWARE (BCD-элемент groupaware), кроме того, заставляет ядро по возможности обойти группу 0, назначая самое большое количество групп, доступное в таких действиях, как выбор потоков и родственности DPC, и обработка назначения групп. Избегайте назначения группе размера 1, поскольку это заставит практически все приложения в системе вести себя так, будто они запущены на однопроцессорной машине, поскольку ядро настраивает маску родственности заданного процесса на охват только одной группы, пока приложение не запросит обратное (что в настоящее время не будет делать большинство приложений).

Следует заметить, что, в крайнем случае, когда число логических процессоров в пакете не может поместиться в одну группу, Windows подгоняет это количество таким образом, чтобы пакет мог поместиться в одной группе, сокращая число CoresPerPhysicalProcessor, а если SMT также не может поместиться, делает то же самое с LogicalProcessorsPerCore. Исключение из этого правила относится к системе, фактически состоящей из нескольких NUMA-узлов в одном пакете. Хотя на момент написания данной книги такой возможности еще не было, в будущем производители процессоров собираются поставлять модули, состоящие из нескольких микросхем — Multiple-Chip Modules (MCM), — представляющие собой расширение многоядерных пакетов. В этих модулях в одном пакете или

на одном кристалле находятся сразу два набора ядер, а также два контроллера памяти. Если в таблице ACPI SRAT определяется наличие MCM-пакета, имеющего два NUMA-узла, то в зависимости от алгоритмов конфигурации групп Windows может связать эти два узла с двумя разными группами. В данном сценарии MCM-пакет будет охватывать более одной группы.

Кроме того, что это дополнительное оборудование становится причиной существенных проблем совместимости драйверов и приложений (которые оно должно по замыслу разработчиков обнаруживать и искоренять), оно оказывает еще большее влияние на машину: оно внедрит NUMA-поведение даже на той машине, которая не обладает NUMA-структурой. Причина в том, что Windows никогда не позволит NUMA-узлу охватывать несколько групп, что следует из алгоритмов назначения. Следовательно, если ядро искусственно создает небольшие группы, у каждой из этих двух групп должен быть свой собственный NUMA-узел. Например, на четырехъядерном процессоре с размером групп равным двум будут созданы две группы и, следовательно, два NUMA-узла, которые станут подузлами основного узла. Это повлияет на политики планирования и управления памятью так же, как это было бы на настоящей NUMA-системе, что может пригодиться при тестировании.

Состояние логического процессора

В дополнение к очередям готовых потоков и к сводке готовности Windows поддерживает две битовых маски, в которых отслеживается состояние процессоров в системе. Перечислим поддерживаемые Windows битовые маски.

Сначала следует упомянуть о маске активных процессоров (KeActiveProcessors), в которой имеется набор битов для каждого используемого в системе процессора. Число наборов битов может быть меньше, чем количество имеющихся процессоров, если согласно лицензионным ограничениям запущенной версией Windows поддерживается меньше процессоров, чем количество доступных физических процессоров. Для проверки нужно воспользоваться переменной KeRegisteredProcessors и посмотреть, сколько процессоров лицензировано на машине. В данном случае под словом «процессор» понимаются физические наборы. С другой стороны, в переменной KeMaximumProcessors содержится максимальное количество логических процессоров, включая все возможные в будущем динамически добавляемые процессоры, которое ограничено лицензионным соглашением, а также любыми ограничениями платформы, запрашиваемыми путем вызова HAL и проверки с использованием таблицы ACPI SRAT, если таковая имеется.

Сводка простоя (KiIdleSummary) является массивом из двух расширенных битовых масок. В первой из них, которая называется CpuSet, каждый набор битов представляет простаивающий процессор, а во втором, SMTSet, каждый бит дает описание простаивающего SMT-набора.

Сводка незапаркованности (KiNonParkedSummary) определяет с помощью установленного бита незапаркованный логический процессор.

Масштабируемость планировщика

Поскольку на мультипроцессорных системах одному процессору может понадобиться изменить структуры данных планирования, исключительно принадле-

жащие другому процессору (например, вставить поток, которому нужно выполняться на конкретном процессоре), доступ к этим структурам синхронизирован с помощью выстраиваемых в очередь спин-блокировок, принадлежащих каждому блоку PRCB, которые удерживаются на уровне диспетчеризации DISPATCH_LEVEL. Таким образом, выбор потока может произойти только при блокировки одного процессорного PRCB-блока. Если нужно, может быть заблокирован еще один процессорный PRCB-блок, например, при сценариях захвата потоков, которые будут рассмотрены позже. Переключение контекста потока также синхронизируется путем использования более тонко устроенной спин-блокировки, относящейся к конкретному потоку.

У каждого центрального процессора имеется также список потоков, находящихся в состоянии отложенной готовности. В нем представлены потоки, которые готовы к запуску, но еще не подготовлены к выполнению; фактически операция приведения их в полную готовность отложена до более подходящего момента. Поскольку каждый процессор управляет только своим собственным списком потоков, находящихся в состоянии отложенной готовности, этот список не синхронизируется спин-блокировкой PRCB-блока. Список потоков, находящихся в состоянии отложенной готовности, обрабатывается функцией KiProcessDeferredReadyList после того, как функция уже внесла изменения в значения родственности процесса или потока, приоритета (включая все, что относится к повышению приоритета) или кванта времени.

Для каждого потока, имеющегося в данном списке, эта функция вызывает функцию KiDeferredReadyThread, которая выполняет алгоритм, показанный далее в разделе «Выбор процессора». Алгоритм может либо вызвать немедленное выполнение потока, помещая его в список готовых потоков процессора, либо, если процессор недоступен, дать потенциальную возможность потоку быть помещенным в список отложенных готовых потоков другого процессора, перейти в состояние повышенной готовности (standby) или немедленно выполниться. Это свойство используется при работе механизма парковки ядра — Core Parking engine: все потоки помещаются в список отложенных готовых потоков, который затем обрабатывается. Поскольку функция KiDeferredReadyThread обходит запаркованные ядра (как будет показано далее), это заставляет все потоки процессора завершать свое выполнение на других процессорах.

Родственность

У каждого потока есть маска родственности, которая указывает на разрешенные для выполнения потока процессоры. Маска родственности потока наследуется от маски родственности процесса. По умолчанию сначала у всех процессов (а следовательно, и у всех потоков) маска родственности эквивалентна набору из всех активных процессоров в назначенной для них группе, иными словами, система может свободно планировать выполнение всех потоков на любом доступном процессоре внутри группы, связанной с процессом.

Но для оптимизации пропускной способности, распределения рабочей нагрузки на определенном наборе процессоров или решения обеих задач приложения могут выбрать изменение для потока маски родственности. Это может быть сделано на нескольких уровнях:

- Путем вызова функции SetThreadAffinityMask для установки родственности для отдельного процесса.
- Путем вызова функции SetProcessAffinityMask для установки родственности всех потоков в процессе. Диспетчер задач (Task Manager) и Process Explorer предоставляют для этой функции GUI, если щелкнуть правой кнопкой мыши на имени процесса и выбрать пункт Задать соответствие (Set Affinity). Инструментальное средство Psexec (от Sysinternals) предоставляет для этой функции интерфейс командной строки. (Обратите внимание на ключ –a в справке по этому средству.)
- Путем включения процесса в состав задания, у которого имеется общая для всего задания маска родственности, установленная с помощью функции SetInformationJobObject.
- Путем задания маски родственности в заголовке образа при компиляции приложения[1].

У образа также может быть во время компоновки установлен флаг однопроцессорности — «uniprocessor». В таком случае во время создания процесса система выбирает один процессор (MmRotatingProcessorNumber), отмечает этот выбор в маске родственности, начиная с первого процессора и затем циклически перебирая все процессоры в группе. Например, на системе с двумя процессорами при первом запуске образа, помеченного как однопроцессорный, этот образ назначается центральному процессору 0, во второй раз — центральному процессору 1, в третий раз — центральному процессору 0, в четвертый раз — центральному процессору 1 и т. д. Этот флаг может пригодиться в качестве временного метода обхода для программ, у которых имеются ошибки многопотоковой синхронизации, которые из-за условий конкуренции проявляются на мультипроцессорных системах, но не случаются на однопроцессорных. Если образ демонстрирует такие симптомы и не имеет цифровой подписи, флаг может быть установлен вручную путем редактирования заголовка образа с помощью такого средства, как Imagecfg.exe. Более удачным решением, к тому же совместимым с исполняемыми файлами, имеющими цифровую подпись, является использование инструментария совместимости приложений Microsoft Application Compatibility Toolkit и добавление прокладки, чтобы заставить базу данных совместимости пометить образ на момент его запуска предназначенным для работы только на однопроцессорной системе.

ЭКСПЕРИМЕНТ: ПРОСМОТР И ИЗМЕНЕНИЕ РОДСТВЕННОСТИ ПРОЦЕССА

В этом эксперименте будут изменены настройки родственности для процесса и отслежено наследование родственности этого процесса новым процессом:

1. Запустите окно командной строки (Cmd.exe).
2. Запустите Диспетчер задач (Task Manager) или Process Explorer и найдите процесс Cmd.exe в списке процессов.

[1] Дополнительные сведения о подробностях формата Windows-образов можно найти на веб-сайте www.microsoft.com по поисковой строке «Portable Executable and Common Object File Format Specification».

3. Щелкните правой кнопкой мыши на имени процесса и выберите пункт Задать соответствие (Set Affinity). Будет выведен список процессоров. Например, на двухпроцессорной системе вы увидите следующее окно.

4. Выберите поднабор доступных в системе процессоров и щелкните на кнопке OK. Теперь потоки процесса могут выполняться только на выбранных вами процессорах.
5. Теперь запустите программу Блокнот (Notepad.exe) из командной строки (набрав в ней Notepad.exe).
6. Вернитесь в Диспетчер задач (Task Manager) или в Process Explorer и найдите новый процесс Notepad. Щелкните на его имени правой кнопкой мыши и выберите пункт Задать соответствие (Set Affinity). Вы увидите тот же самый перечень процессоров, который был выбран для процесса окна командной строки. Дело в том, что процессы наследуют настройки родственности от своих родителей. ■

Windows не будет перемещать выполняемый поток, который может выполняться на другом процессоре, с одного центрального процессора на другой процессор, чтобы разрешить потоку с родственностью, связанной с первым процессором, выполняться на первом процессоре. Рассмотрим, например, следующий сценарий: центральный процессор 0 выполняет поток с уровнем приоритета 8, который может выполняться на любом процессоре, а центральный процессор 1 выполняет поток с уровнем приоритета 4, который тоже может выполняться на любом процессоре. Становится готовым к выполнению поток с уровнем приоритета 6, который может выполняться только на центральном процессоре 0. Что при этом произойдет? Windows не станет перемещать поток с уровнем приоритета 8 с центрального процессора 0 на центральный процессор 1 (вытесняя тем самым поток с уровнем приоритета 4), так что готовый к выполнению поток с уровнем приоритета 6 должен оставаться в состоянии готовности.

Поэтому изменение маски родственности процесса или потока может привести к тому, что потоки получат меньше времени центрального процессора, чем обычно, потому что Windows ограничена в запуске потока на конкретном процессоре. Следовательно, настройка родственности должна осуществляться с особой осмотрительностью, в большинстве случаев лучше всего позволить системе Windows решать, какой поток где запускать.

Расширенная маска родственности

Для поддержки более 64 процессоров, преодолевая ограничение, накладываемое структурой маски родственности (состоящей из 64 битов на 64-разрядной системе), Windows использует расширенную маску родственности (KAFFINITY_EX), представляющую собой массив из масок родственности, по одной для каждой поддерживаемой группы процессоров (на данный момент их число равно 4). Когда планировщику нужно сослаться на процессор в расширенных масках

родственности, он сначала выделяет из ссылки нужную битовую маску, используя ее номер группы, а затем напрямую обращается к полученным данным о родственности. В API-функциях ядра расширенные маски родственности не фигурируют, вместо этого код, вызывающий API-функцию, вводит номер группы в виде аргумента и получает обычную маску родственности для этой группы. Но с другой стороны, у API-функций Windows обычно можно запросить только информацию об одной группе, которая является группой текущего выполняемого потока (которая имеет фиксированное значение).

Расширенная маска родственности и ее основные функциональные возможности определяют также и способ преодоления процессом границ исходной назначенной группы процессоров. Путем использования API-функций расширенной маски родственности потоки в процессе могут выбрать маски родственности на другой группе процессоров. Например, если у процесса есть 4 потока и у машины имеется 256 процессоров, поток 1 может выполняться на процессоре 4, поток 2 может выполняться на процессоре 68, поток 4 на процессоре 132, а поток 4 на процессоре 196, если для каждого потока будет установлена маска родственности `0x10` (`0b10000` в двоичном формате) на группах 0, 1, 2 и 3. В качестве альтернативы для каждого потока может быть установлена маска родственности `0xFFFFFFFF` для их исходных групп, и процесс может выполнять свои потоки на любом доступном процессоре системы (с тем лишь ограничением, что каждый поток выполняется только внутри своей собственной группы).

Использоваться расширенная маска родственности должна во время создания, путем указания номера группы при создании нового потока в списке его атрибутов. (Дополнительные сведения о списках атрибутов даны в предыдущей теме, касавшейся создания потоков.)

Маска родственности системы

Поскольку драйверы Windows обычно выполняются в контексте вызывающих потоков или в контексте произвольного потока (то есть вне границ безопасности), текущий выполняемый код драйвера может быть субъектом правил родственности, установленных разработчиком приложения, которые в настоящее время не соответствуют коду драйвера и могут даже помешать правильной обработке прерываний и другой очередной работе. Поэтому у разработчиков драйверов имеется механизм временного обхода настроек родственности пользовательского потока путем использования API-функций `KeSetSystemAffinityThread(Ex)`/`KeSetSystemGroupAffinityThread` и `KeRevertToUserAffinityThread(Ex)`/`KeRevertToUserGroupAffinityThread`.

Идеальный и последний процессор

У каждого потока есть три номера центрального процессора, хранящиеся в блоке управления процессом, находящемся в ядре:

- идеальный, или предпочтительный процессор, на котором должен выполняться этот поток;
- последний процессор, или процессор, на котором поток выполнялся в последний раз;

- следующий процессор, или процессор, на котором поток будет выполняться или уже выполняется.

Идеальный процессор выбирается для потока при создании этого потока с использованием источника (seed) в блоке управления процессом. Источник получает приращение при каждом создании потока, поэтому идеальный процессор для каждого нового потока в процессе выбирается по кругу среди процессоров, доступных в системе. Например, первому потоку в первом процессе системы назначается идеальный процессор 0. Второму потоку в этом процессе назначается идеальный процессор 1. Но следующий процесс в системе назначает своему первому потоку идеальный процессор 1, второй 2 и т. д. Таким образом, потоки каждого процесса распространяются среди процессоров.

Следует заметить, что при таких назначениях потоков внутри процессов предполагается, что у них будет одинаковый объем работы. Обычно такое в многопотоковых процессах не случается, потому что есть один или несколько служебных потоков, а также несколько рабочих потоков. Поэтому многопотоковое приложение, которое собирается всецело воспользоваться платформой, должно находить преимущества в указании номеров идеальных процессоров для своих потоков путем использования функции SetThreadIdealProcessor. Чтобы получить преимущество от групп процессоров, разработчики должны вместо этого вызывать функцию SetThreadIdealProcessorEx, которая позволяет выбирать номер группы для указания родственности.

На 64-разрядной версии Windows для соблюдения баланса назначений вновь создаваемым потокам в процессе в блоке KPRCB используется поле прогресса Stride. Это поле является скалярным числом, представляющим количество битов родственности в заданном NUMA-узле, которое должно быть пропущено для получения новой независимой логической части процессора. Слово «независимой» означает «расположенной на другом ядре» (для SMT-системы) или «на другом наборе» (для многоядерной системы без использования SMT-технологии). Поскольку 32-разрядная версия Windows не поддерживает системы с большими процессорными конфигурациями, в ней поле stride не используется, и она просто выбирает номер следующего процессора, пытаясь по возможности избежать совместного использования одного и того же SMT-набора. Например, на двухпроцессорной SMT-системе с четырьмя логическими процессорами, если в качестве идеального процессора для первого потока назначен логический процессор 0, второму потоку будет назначен логический процессор 2, третьему потоку будет назначен логический процессор 1, а четвертому потоку будет назначен логический процессор 3 и т. д. Таким образом, потоки равномерно распространяются по физическим процессорам.

Идеальный узел

При создании процесса на NUMA-системах для него выбирается идеальный узел. Первый процесс назначается узлу 0, второй процесс узлу 1 и т. д. Затем из идеального узла процесса для потоков процесса выбираются идеальные процессоры. В качестве идеального процессора для первого потока в процессе назначается первый процессор в узле. По мере создания в процессе дополнительных потоков с тем же самым идеальным узлом для следующего потока в качестве идеального используется следующий процессор и т. д.

Выбор потока на мультипроцессорных системах

Перед более подробным рассмотрением мультипроцессорных систем подведем краткий итог по алгоритмам, рассмотренным в разделе «Выбор потока». Согласно этим алгоритмам либо продолжал выполняться текущий поток (если не был найден новый кандидат), либо запускался поток простоя (если текущий поток блокировался). Но есть еще и третий алгоритм выбора потока, упомянутый в ранее рассмотренном разделе «Планировщик простоя», который реализован в функции KiSearchForNewThread. Этот алгоритм вызывается в одном конкретном случае: когда текущий поток близок к блокировке из-за ожидания объекта, включая ситуацию вызова функции NtDelayExecutionThread, которая так же известна в Windows, как API-функция спячки — Sleep API.

ПРИМЕЧАНИЕ

Между часто используемым вызовом Sleep(1), вызывающим блокировку текущего потока до следующего такта таймера, и показанным ранее вызовом SwitchToThread() есть небольшая разница. При вызове «спячки» (sleep) будет использоваться алгоритм, который вскоре будет рассмотрен, в то время как при вызове «уступки» (yield) используется ранее рассмотренная логика.

Функция KiSearchForNewThread сначала проверяет, был ли поток уже выбран для выполнения на этом процессоре (путем чтения поля NextThread); если был, то тут же происходит диспетчеризация этого потока в состояние выполняемого (Running). В противном случае вызывается процедура выбора готового потока KiSelectReadyThread, и если поток был найден, выполняются такие же действия.

Если же поток не был найден, процессор помечается простаивающим (даже если поток простоя еще не выполняется) и инициируется сканирование очередей другого логического процессора (в отличие от других стандартных алгоритмов, которые теперь отказались бы от дальнейших попыток). Кроме того, поскольку процессор теперь считается простаивающим, если включен режим распределенного справедливого долевого планирования — Distributed Fair Share Scheduling, — то поток, по возможности, будет освобожден из очереди простоя (idle-only queue), и его выполнение будет перепланировано. С другой стороны, если ядро процессора в данный момент запарковано, алгоритм не будет пытаться проверить другие логические процессоры, поскольку предпочтительнее позволить ядру войти в состояние парковки, нежели держать его загруженным новой работой.

Исключая эти два сценария, теперь запускается цикл захвата работы. Код этого цикла смотрит на текущий NUMA-узел и удаляет все простаивающие процессоры, поскольку они не могут иметь потоков, нуждающихся в захвате. Затем, начиная с процессора с самым большим номером, цикл вызывает функцию поиска готового потока KiFindReadyThread, но указывает ей удаленный, а не текущий блок KPRCB, заставляя этот процессор искать наиболее подходящий готовый поток из очереди другого процессора. Если ничего не будет найдено и включен планировщик распределенного справедливого долевого планирования — Distributed Fair Share Scheduler, — то, по возможности, поток, который был в очереди простоя удаленного логического процессора, освобождается на текущем процессоре.

Если кандидаты из готовых потоков найдены не были, попытка повторяется в отношении следующего логического процессора с меньшим номером и т. д., пока не закончатся все логические процессоры текущего NUMA-узла. В таком случае алгоритм продолжит поиск в следующем ближайшем узле и т. д., пока не закончатся все узлы в текущей группе (Windows позволяет отдельно взятому потоку иметь родственную связь только с одной группой). Если не будет найден ни один кандидат, функция возвращает NULL, и процессор в случае ожидания переходит к выполнению потока простоя (при этом планирование простоя пропускается). Если эта работа уже была выполнена планировщиком простоя, процессор входит в состояние спячки.

Выбор процессора

До сих пор мы рассматривали те способы, которые используются Windows при выборе потока, когда такой выбор нужен логическому процессору (или должен быть сделан заданным логическим процессором), и предполагали, что у различных процедур планирования есть существующая база данных готовых потоков, из которых можно сделать этот выбор. Теперь мы первым делом рассмотрим, как эта база данных заполняется — иными словами, как Windows решает вопрос следующего выбора: с очередью готовых потоков какого логического процессора должен быть связан заданный готовый поток. После того как были рассмотрены типы мультипроцессорных систем, поддерживаемых Windows, а также родственность потоков и назначение идеального процессора, теперь можно рассмотреть порядок использования этой информации для выбора процессора.

Выбор процессора для потока при наличии простаивающих процессоров

Когда поток становится готовым к выполнению, вызывается функция планировщика KiDeferredReadyThread, заставляя Windows выполнять две задачи: настроить, если нужно, приоритеты и обновить кванты, что уже было рассмотрено в разделе «Повышение приоритета», а затем выбрать для потока наиболее подходящий логический процессор.

Сначала Windows ищет для потока идеальный процессор, а затем вычисляет набор идеальных процессоров исходя из принадлежащей потоку жестко заданной маски родственности. Затем этот набор сокращается следующим образом:

1. Удаляются любые простаивающие логические процессоры, запаркованные с помощью механизма парковки ядра — Core Parking. При отсутствии простаивающих процессоров выбор таких процессоров прекращается, и планировщик ведет себя как при отсутствии доступных простаивающих процессоров.
2. Удаляются любые простаивающие логические процессоры, не относящиеся к идеальному узлу (определенном как узел, содержащий идеальный процессор), если только это не вызовет исключения всех простаивающих процессоров.
3. На SMT-системах удаляются любые непростаивающие SMT-наборы, даже если это может вызвать исключение именно того процессора, который считался идеальным. Иными словами, Windows отдает приоритет неидеальному, простаивающему SMT-набору перед идеальным процессором.

4. Затем Windows проверяет, не находится ли идеальный процессор среди оставшихся наборов простаивающих процессоров. Если нет, она должна затем найти наиболее подходящий простаивающий процессор. Это делается путем предварительной проверки, не является ли процессор, на котором поток выполнялся в последний раз, частью набора оставшихся простаивающих процессоров. Если да, то этот процессор рассматривается как временный идеальный процессор, и выбор останавливается на этом процессоре. (Следует напомнить, что идеальный процессор пытается максимизировать число попаданий в кэш-память процессора, и выбор последнего процессора, на котором выполнялся поток, является неплохим способом добиться этого результата.)
5. Если последний процессор не является частью набора оставшихся простаивающих процессоров, Windows проверяет, не является ли частью этого набора текущий процессор (то есть процессор, который в данный момент выполняет планирующий код). Если он относится к этому набору, то к нему применяется та же самая логика, которая была описана в предыдущем пункте.
6. Если к простаивающим не относится ни последний, ни текущий процессор, Windows выполняет еще одну операцию сокращения, удаляя любой простаивающий логический процессор, не входящий в тот же SMT-набор, что и идеальный процессор. Если таких процессоров уже не осталось, Windows вместо этого удаляет любые процессоры, не входящие в SMT-набор текущего процессора, кроме этого, удаляет также все простаивающие процессоры. Иными словами, Windows предпочитает простаивающие процессоры, которые совместно используют тот же самый SMT-набор, что и недоступный идеальный процессор и (или) последний процессор, который было бы предпочтительнее выбрать в первую очередь. Поскольку реализации SMT совместно используют кэш-память, имеющуюся в ядре, это будет иметь с точки зрения рационального использования кэш-памяти примерно тот же эффект, что и выбор идеального или последнего процессора.
7. И наконец, если в последнем пункте в идеальном наборе останется более одного процессора, Windows выберет в качестве текущего процессора потока процессор с меньшим номером.

После выбора процессора, на котором будет выполняться поток, этот поток переводится в состояние повышенной готовности (standby), и PRCB-блок простаивающего процессора обновляется, чтобы указывать на этот поток. Если процессор простаивает, но не остановлен, ему отправляется DPC-прерывание, чтобы процессор тут же приступил к операции планирования.

Как только такая операция планирования инициируется, вызывается функция KiCheckForThreadDispatch, которая установит, что для выполнения на процессоре был спланирован новый поток, что немедленно станет причиной контекстного переключения, если это возможно (а также доставки отложенных APC-вызовов), или же это станет причиной отправки DPC-прерывания.

Выбор процессора для потока при отсутствии простаивающих процессоров

Когда при необходимости выполнения потока простаивающие процессоры отсутствуют, или если при первом сокращении был удален только один простаиваю-

щий процессор (при избавлении от запаркованных простаивающих процессоров), Windows сначала проверяет, не сложилась ли последняя из этих ситуаций. При таком сценарии планировщик вызывает функцию KiSelectCandidateProcessor, чтобы спросить у механизма парковки ядра Core Parking, какой из процессов будет лучшим кандидатом. Core Parking выбирает процессор с наибольшим номером, который незапаркован на простаивающем идеальном узле. Если таких процессоров нет, механизм принудительно отменяет состояние парковки идеального процессора и заставляет его распарковаться.

По возвращении в планировщик будет проведена проверка, является ли полученный кандидат простаивающим, и если да, планировщик выберет этот процессор для потока, следуя таким же последним шагам, как и в предыдущем сценарии.

При неудаче Windows сравнивает приоритет потока, выполняемого (или находящегося в состоянии повышенной готовности) на процессоре, со считающимся идеальным, чтобы определить, должен ли он вытеснить этот поток.

Если идеальный для потока процессор уже имеет поток, выбранный в качестве следующего потока для выполнения (ожидающего в состоянии повышенной готовности дальнейшего планирования), и уровень приоритета этого потока ниже уровня приоритета потока, подготовленного к выполнению, новый поток вытесняет первый поток из состояния повышенной готовности и становится следующим потоком для этого центрального процессора. Если на этом центральном процессоре уже есть выполняемый поток, Windows проверяет, не имеет ли выполняемый поток уровень приоритета ниже уровня приоритета потока, подготовленного к выполнению. Если так есть, текущий выполняемый поток помечается на вытеснение, и Windows ставит в очередь DPC-прерывание на целевой процессор для вытеснения текущего выполняемого потока в пользу нашего нового потока.

Если готовый поток не может быть запущен немедленно, он переводится в состояние готового потока и помещается в очередь потоков того уровня приоритета, который соответствует уровню приоритета этого потока, где он будет дожидаться своей очереди на запуск. Как было показано в ранее рассмотренных сценариях планирования, поток будет вставлен либо в начало, либо в конец очереди, в зависимости от того, входил ли он в состояние готовности из-за вытеснения.

Таким образом, независимо от положенного в основу сценария и различных возможностей, следует иметь в виду, что потоки всегда помещаются в индивидуальные очереди готовности своих идеальных процессоров, что гарантирует постоянство алгоритмов, определяющих, как логический процессор выбирает поток для выполнения.

Планирование, основанное на долевом использовании процессора

В предыдущем разделе рассматривалась реализация в Windows стандартного планирования на основе потоков, которая надежно служила общим пользовательским и серверным сценариям с момента своего появления в первом выпуске Windows NT (с усовершенствованиями, касающимися масштабирования, вносимыми с каждым выпуском). Планирование, основанное на потоках, предпринимает

попытки справедливо распределять время процессора или процессоров только среди конкурирующих потоков с одинаковым уровнем приоритета. Поэтому в нем не берутся в расчет такие высокоуровневые требования, как распределения потоков по пользователям и потенциал получения конкретными пользователями преимуществ от большего объема общего времени центрального процессора за счет других пользователей. Как выясняется, такой тип поведения высоко ценится в среде терминальных служб, где за получение времени центрального процессора соревнуются десятки пользователей, и если используется только планирование на основе потоков, один высокоприоритетный поток от того или иного пользователя имеет потенциальную возможность посадить на голодный паек потоки от остальных пользователей.

Распределенное справедливое долевое планирование

В этом разделе будут рассмотрены два альтернативных режима планирования, реализованных в последних версиях Windows: основанное на сеансах распределенное справедливое долевое планирование — session-based Distributed Fair Share Scheduler (DFSS) — и более старая, традиционная реализация ограничения частоты процессора на основе SID.

Инициализация DFSS

В ходе самой последней части инициализации системы, когда Smss инициализирует SOFTWARE-кучу, диспетчер процессов инициирует в функции PsBootPhaseComplete последнюю инициализацию после загрузки, которая вызывает функцию PsInitializeCpuQuota. Именно здесь система принимает решение, какой из двух механизмов квот центрального процессора (DFSS или традиционный) будет задействован. Чтобы включить DFSS, должно быть установлено в единицу значение параметра реестра EnableCpuQuota в обеих разделах квот: HKLM\SOFTWARE\Policies\Microsoft\Windows\Session Manager\QuotaSystem для настроек, основанных на политике[1], а также в системном разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Quota System (по умолчанию установлено в TRUE), в котором определяется, поддерживает ли система эту функциональную возможность.

ПРИМЕЧАНИЕ

Из-за ошибки (подробности которой можно найти на ресурсе http://technet.microsoft.com/en-us/library/ee808941(WS.10).aspx) настройки групповой политики по отключению DFSS не соблюдаются. Системные настройки должны быть отключены вручную.

[1] Она может быть сконфигурирована через редактор групповой политики в разделе Конфигурация компьютера (Computer Configuration) ▸ Административные шаблоны (Administrative Templates) ▸ Компоненты Windows (Windows Components) ▸ Служба удаленных рабочих столов (Remote Desktop Services) ▸ Узел сеансов удаленных рабочих столов (Remote Desktop Session Host) ▸ Подключения (Connections). Отключение справедливого долевого планирования центрального процессора — Turn off Fair Share CPU Scheduling.

Если планирование DFSS включено, значение переменной PsCpuFairShareEnabled установлено в true, что заставит ядро посредством различных путей кода планирования вести себя по-другому и (или) обратиться к механизму DFSS. В дополнение к этому квота по умолчанию для каждого DFSS-цикла устанавливается в 150 миллисекунд, на величину, которая называется кредитом (credit).

Как только будет включено DFSS-планирование, для обслуживания его информации, например списка блоков квот центрального процессора для каждого сеанса (а также спин-блокировок и подсчетов) и общего веса всех сеансов системы, используется глобальная структура данных PspCpuQuotaControl. Также там хранится массив структур данных DFSS для каждого процессора, который будет показан далее.

Блоки квот центрального процессора для каждого сеанса

После включения DFSS при каждом создании нового сеанса (отличающегося от сеанса 0) функция MiSessionCreate вызывает функцию PsAllocateCpuQuotaBlock для настройки блока квот центрального процессора для этого сеанса. Когда это происходит на системе в первый раз (например, для сеанса 1), для завершения инициализации DFSS вызывается функция PspLazyInitializeCpuQuota.

В результате этого выделяется упомянутая в предыдущем разделе структура данных DFSS, создаваемая для каждого центрального процессора, которая содержит DPC-вызов, используемый для управления квотой (показанной далее процедуры PspCpuQuotaDpcRoutine), и общее количество кредитуемых, а также аккумулируемых циклов. В этой структуре также хранится блочное поколение, монотонно наращиваемая последовательность, обеспечивающая гарантию атомарности, а также хранится блокировка очереди простоя, защищающая список одинаковых имен, который является центральным элементом еще не рассмотренного механизма DFSS. Каждая структура данных DFSS, принадлежащая отдельному процессору, в свою очередь, связывается посредством отсортированного списка с двойной связью с различными блоками квот центрального процессора, принадлежащими каждому сеансу, которые упоминались в начале этого рассмотрения.

Когда первоначальная инициализация DFSS завершается, функция PsAllocateCpuQuotaBlock может продолжить свою работу, первым делом выделяя для данного сеанса блок квоты центрального процессора. Эта структура поддерживает общую учетную информацию сеанса, а также результаты отслеживания каждого центрального процессора, включая количество оставшихся и первоначально выделенных циклов, и саму очередь простоя в структуре записей квот каждого центрального процессора.

Сначала сохраняется идентификатор сеанса (session ID), и общему весу центрального процессора устанавливается его значение по умолчанию равное 5. Вскоре будет показано, что такое вес, как он может быть вычислен и каково его влияние на механизм DFSS. Поскольку блок квоты был только что создан, все исходные значения циклов пока установлены на их максимальные величины. Затем этот блок центрального процессора для каждого сеанса должен быть видимым в системе. Поэтому структура данных PspCpuQuotaControl получает новый общий вес всех сеансов (путем сложения этого веса), и блок квоты вставляется в список блоков (отсортированный по идентификаторам сеансов). И наконец,

функция PspCalculateCpuQuotaBlockCycleCredits пересчитывает блок квоты каждого другого сеанса и получает новый общий вес системы.

Как только это будет сделано, блок квоты центрального процессора для каждого сеанса получает окончательное значение, и диспетчер памяти помещает его значение в поле CpuQuotaBlock структуры MM_SESSION_SPACE для данного сеанса. Точно так же обновляется и EPROCESS (часть этого поля CpuQuotaBlock нового сеанса), чтобы указывать на блок квоты центрального процессора этого сеанса. Теперь, когда процесс получил блок квоты центрального процессора, и как только он стал частью сеанса, будущим потокам, создаваемым этим процессом (включая и сам первый поток), будет выделяться дополнительная структура после их обычной структуры ETHREAD — APC-структура квоты центрального процессора для каждого процессора. Кроме того, полю RateApcState структуры ETHREAD будет дано значение PsRateApcContained, показывающее, что это встроенный APC-вызов квоты (Quota APC), используемый механизмом DFSS (а не традиционный APC-вызов, выделяющий пул). И наконец, в KTHREAD-переменной ThreadControlFlags устанавливается бит дросселирования центрального процессора — CpuThrottled.

К этому моменту в глобальной структуре управления квотами содержится указатель на массив структуры данных DFSS для каждого центрального процессора, который сам связан со всеми блоками центрального процессора, которые создавались для каждого сеанса, и связан со структурой EPROCESS участвующих процессов. В свою очередь, каждый поток, являющийся частью такого процесса, имеет включенное дросселирование центрального процессора. Есть готовый к выполнению DPC-вызов для каждого центрального процессора, а также APC-вызовы для каждого дросселируемого потока.

Когда последний процесс сеанса теряет все свои ссылки, вызывается функция PsDeleteCpuQuotaBlock. Она удаляет блок из списка, обновляет значение общего веса и вызывает функцию PspCalculateCpuQuotaBlockCycleCredits для обновления всех других блоков квот центрального процессора для каждого сеанса.

Зарядка циклов дросселируемых потоков

После того как все настроено, весь DFSS-механизм дросселируется с помощью расхода циклов центрального процессора, о чем уже говорилось в предыдущих разделах. Иными словами, расходуемые циклы используются не только для подсчета квантов и предоставления точной информации API-функциям потока, но они также могут быть «заряжены» в потоке (а значит, и в его квоте). Эта операция осуществляется функцией PsChargeProcessCpuCycles, которая вызывается, когда поток завершил аккумуляцию циклов во время своего текущего выполнения.

Первая операция включает в себя аккумулирование дополнительных циклов в структуре данных DFSS данного процессора, которая имеется у каждого центрального процессора, увеличивая значение TotalCyclesAccumulated. Если это аккумулирование достигнет полного кредита, тут же будет поставлен в очередь DPC-вызов квоты. Когда по прошествии некоторого времени будет выполняться DPC-вызов, будет вызвана функция PspStartNewFairShareInterval, которая обновит поколение, сбросит аккумулированные циклы и переустановит кредит на 150 мс. И наконец, на каждом процессоре, связанным с данным сеансом, будет

очищена очередь простоя. Эта часть алгоритма управляет 150 мс интервалом, который регулирует работу DFSS.

Вторая возможность обуславливается тем, что поколение (generation) записи квоты центрального процессора, содержащееся в блоке квот центрального процессора текущего процесса (принадлежащего сеансу), не управляет поколением структурой данных DFSS каждого процессора. Это несовпадение поколений подсказывает, что был достигнут новый интервал и что пока еще не был установлен новый лимит циклов, поэтому вызывается функция PspReplenishCycleCredit, чтобы проделать эту работу. Считывается вес каждого центрального процессора и общий вес, который был ранее помещен в переменную PspCalculateCpuQuotaBlockCycleCredits, и эти данные используются для установки базовой нормы циклов для текущих данных каждого процессора внутри имеющегося у процесса блока квот центрального процессора. Для этого используется простая формула: процесс получает эквивалент его кредита циклов (150 мс), поделенный на общий вес всех сеансов в системе. Затем количество циклов, в течение которых ему разрешено выполняться (CyclesRemaining), устанавливается равным произведению базовой норме циклов и весом данного конкретного сеанса. Иными словами, процесс выполняется в течение полученной в результате справедливой дележки части времени с учетом количества других сеансов, вычисленной в виде процента на основе относительного веса данного сеанса в сравнении с общим весом всей системы. Когда вычисление будет завершено, поколения устанавливаются соответствующими друг другу.

Во всех остальных случаях функция PsChargeProcessCpuCycles просто вычитает количество циклов из значения переменной CyclesRemaining, а затем вызывает функцию PsCheckThreadCpuQuota, чтобы посмотреть, не было ли это количество циклов исчерпано (достигнув нуля). Следует заметить, что иногда эта функция может также быть вызвана непосредственно из кода переключения контекста, когда управление вот-вот будет передано потоку, для которого включено дросселирование центрального процессора.

Функция PsCheckThreadCpuQuota восстанавливает блок квоты центрального процессора для данного процесса (то есть для сеанса), а затем дополнительно извлекает из него точную информацию, относящуюся к центральному процессору. Она еще раз проверяет несовпадение поколений, что будет свидетельствовать, что это первая зарядка для этого 150 мс кредита циклов, а затем она вызывает функцию PspReplenishCycleCredit. После этого проводится проверка, не показывает ли блок квоты центрального процессора для данного процесса, что циклов больше не осталось. Если циклы еще остались, функция возвращает управление, в противном случае она готовится к приостановке выполнения потока.

Перед остановкой выполнения функция извлекает DPC центрального процессора, убеждаясь в том, что он (или связанный с ним APC потока) еще не выполняется. Если эта операция выполняется из-за ранее запущенного сценария переключения контекста, APC потока помещается в очередь, что вызовет вытеснение выполнения потока, как только будет завершено переключение контекста. В противном случае, если причиной выполнения операции будет зарядка циклов (происходящая на уровне не ниже DISPATCH_LEVEL), то вместо этого в очередь помещается DPC центрального процессора, что впоследствии приведет к помещению в очередь APC потока. (Это вызывает практически немедленный

ответ на ограничение квоты центрального процессора.) В случае, когда последующее аккумулирование циклов пройдет 150 мс кредит циклов, DPC-вызов также вызывает ранее рассмотренную функцию PspStartNewFairShareInterval.

Дросселирование центрального процессора и принудительное задание квот

Итак, вы узнали, как проходит инициализация DFSS, как для каждого сеанса создаются блоки квот центрального процессора (и как они затем связываются с участвующими в сеансах процессами) и как потоки выполняются с установленным битом дросселирования центрального процессора (подразумевает их принадлежность к процессам, участвующим в сеансе при включенном механизме DFSS), потребляя циклы из своей общей, связанной с весом нормой, переустанавливаемой каждые 150 мс. Вы также увидели, как в конечном итоге помещается в очередь APC во всех случаях, когда поток израсходовал положенные ему циклы. А теперь вы поймете, как APC навязывает ограничение квоты центрального процессора.

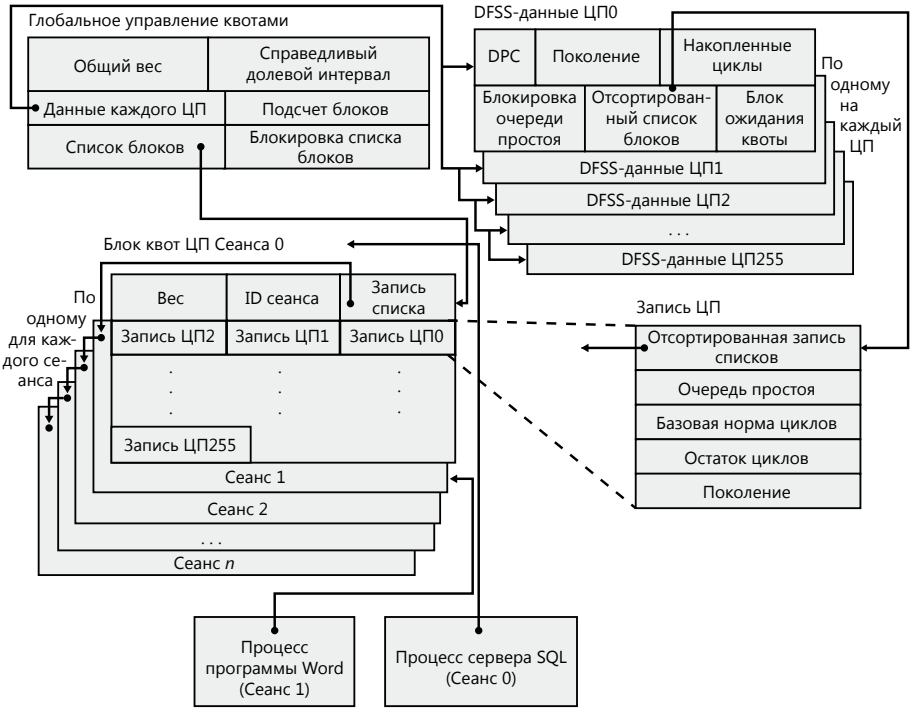
Сначала APC входит в бесконечный цикл, создавая размещаемый в стеке блок ожидания квоты (Quota Wait Block), содержащий ограничиваемый текущий поток, а также выдавая событие. Это именно то событие, которое в конечном итоге позволяет потоку продолжить выполнение. Затем APC-вызов получает указатель на структуру данных DFSS центрального процессора и получает ранее упомянутую блокировку очереди простоя. Затем он проверяет, пуста ли у текущего процессора очередь простоя (которая берется из записи квоты центрального процессора, содержащейся в блоке квот центрального процессора, принадлежащем процессу). Если список пуст, предполагается, что этот центральный процессор никогда не вставлялся в отсортированный список блоков, который содержится в структуре данных DFSS каждого процессора (в части глобального массива PspCpuQuotaControl). В таком случае для исправления ситуации вызывается функция PspInsertQuotaBlockCpuEntry.

Поскольку эту структуру данных использует сам планировщик DFSS (который еще не был рассмотрен), вставка должна быть произведена самым оптимальным образом, в данном случае отсортированной по базовой норме циклов из данных каждого центрального процессора, содержащейся в принадлежащей каждому процессу блоке квот центрального процессора. Следует напомнить, что базовая норма циклов изначально определяется кредитом циклов 150 мс, поделенным на общий вес системы (то есть речь идет о полной норме), но вы еще увидите, как планировщик DFSS может позже изменить норму.

Далее, так как теперь имеющаяся для каждого процессора запись квоты (Quota Entry) находится в отсортированном списке блоков (или могла быть там, если очередь простоя была не пуста), этот поток вставляется в конец очереди простоя и привязывается с помощью записи списка со связью, имеющегося в блоке ожидания квоты (Quota Wait Block). Поскольку этот блок ожидания содержит ранее проинициализированное событие возобновления, DFSS-планировщик может по необходимости управлять потоком.

И наконец, APC входит в ожидание этого события возобновления с причиной ожидания WrCpuRateControl. Используя такой инструментарий, как входящее в комплект Sysinternals средство PsList, или Process Explorer, — каждое из этих

средств показывает (а также отладчик ядра) причины ожидания — вы можете увидеть, какие потоки периодически блокируются на DFSS-системе.



Возобновление выполнения

Со временем, вероятно, достигать своих ограничений, связанных с квотой времени центрального процессора, и блокироваться на своих соответствующих очередях простоя будет все больше и больше потоков, а как же они, в конечном счете, будут возобновлять свое выполнение? Одна из возможностей заключается в запуске нового интервала 150 мс. Если вспомнить ранее рассмотренный материал, то функции PspStartNewFairShareInterval предписывалось «очистить очередь простоя». Эта операция, выполняемая функцией PspFlushProcessorIdleOnlyQueue, по сути, сканирует каждую запись квоты данного процессора (такая запись имеется для каждого процессора и находится в отсортированном списке блоков), а затем сканирует очередь простоя каждого такого процессора. Выбирая каждый поток в списке, функция удаляет поток и самостоятельно запускает событие возобновления. Таким образом, любой заблокированный поток на текущем центральном процессоре получает возможность возобновления выполнения после 150 мс.

Понятно, что очищение не является обычным механизмом, посредством которого управляются потоки очереди простоя. Эта работа обычно выполняется самим DFSS-планировщиком, предоставляющим процедуру PsReleaseThreadFromIdleOnlyQueue в виде функции обратного вызова, которая может использоваться обычным планировщиком потоков, когда система близка к переходу в состояние простоя и когда требуется работа, связанная с DFSS. Конкретно DFSS

вызывается подробно рассмотренной ранее функцией KiSearchForNewThread при следующих двух вариантах развития событий:

- Если изначально вызванная функция выбора готового потока KiSelectReadyThread не нашла новый поток для текущего процессора до того, как она проверила очереди диспетчеров готовых потоков других процессоров, KiSearchForNewThread потребует у DFSS освобождения потока из очереди простоя.
- В противном случае сканируются очереди готовых потоков диспетчера каждого центрального процессора (путем циклического вызова функции KiSelectReadyThread в отношении каждого блока PRCB), если опять-таки поток не будет найден, для освобождения потока из очереди простоя на целевом процессоре также вызывается DFSS-планировщик.

И наконец, вы еще увидите, чем на самом деле занимается функция PsReleaseThreadFromIdleOnlyQueue и как реализован DFSS-планировщик.

DFSS-планирование очереди простоя

Функция освобождения потока из очереди простоя — PsReleaseThreadFromIdleOnlyQueue сначала проверяет, не пуст ли отсортированный список блоков (что будет означать отсутствие даже любых допустимых записей квот для каждого процессора), и если так оно и есть, передает управление вызывавшему коду. В противном случае она приобретает спин-блокировку очереди простоя из структуры данных DFSS, созданной для каждого центрального процессора, и вызывает функцию поиска для выполнения потока с наивысшим приоритетом — PspFindHighestPriorityThreadToRun. Эта функция сканирует отсортированный список блоков, восстанавливая запись квот для каждого центрального процессора, а затем сканирует каждую запись (которая, если помните, указывает на блок ожидания квоты — Quota Wait Block для потока). К сожалению, поскольку потоки не вставлены по приоритетам (как в настоящих диспетчерских очередях готовых потоков), должна быть просканирована вся очередь простоя, и при каждом проходе должен быть записан поток с наивысшим на данный момент приоритетом. (По причине приобретения блокировки, в ходе сканирования не могут вставляться ни новые записи квот для каждого центрального процессора, ни потоки в очередь простоя.)

ПРИМЕЧАНИЕ

Поскольку механизм DFSS не интегрирован по-настоящему с обычным планировщиком потоков, причина того, что потоки не отсортированы по приоритетам, вполне очевидна: механизм DFSS не осведомлен об изменениях приоритетов после того, как в его списки были вставлены потоки из очереди простоя. Пользователь может по-прежнему изменять приоритет, и поскольку планировщик потоков не уведомляет об этом DFSS, будет выбран не тот поток, который следовало бы выбрать.

Кроме того, тщательно проверяется родственность, чтобы гарантировать, что сканируются только потоки соответствующей родственности. Хотя в каждой очереди простоя содержатся только потоки для текущего процессора, во втором варианте развития событий из предыдущего раздела показано, как может быть также просканирована очередь простоя удаленного процессора. Механизм DFSS

должен обеспечить, что на текущем центральном процессоре будет запущен поток из очереди простоя соответствующего удаленного центрального процессора.

После того как поток с наивысшим приоритетом относительно текущей записи квот центрального процессора был найден, он удаляется из очереди простоя и возвращается вызывавшему функцию коду. Кроме того, если это был последний поток в очереди простоя, запись, имеющаяся для каждого процессора, удаляется из отсортированного списка блоков. Поэтому следует заметить, что другие записи квот, имеющиеся для каждого процессора, не проверяются, если только относительно первой записи квоты центрального процессора (то есть того центрального процессора, у которого будет наивысшая базовая норма циклов) не будет найден готовый к выполнению поток с наивысшим приоритетом.

Как только поток будет найден, функция освобождения потока из очереди простоя — PsReleaseThreadFromIdleOnlyQueue — возобновляет его выполнение и еще раз помещает в очередь DPC-вызов, отвечающий за запуск в конечном итоге принадлежащего предыдущему потоку APC-вызова (после того как убедится, что DPC-вызов не был уже запущен). Таким образом, в данном случае APC никогда не будет поставлен в очередь сам по себе, поскольку эта функция выполняется, как часть диспетчера потоков, и уже на уровне DISPATCH_LEVEL. Кроме того, не имело бы смысла ставить в очередь еще один APC, создаваемый для каждого потока, только для того, чтобы уведомить исходный APC; вместо этого поток будет пробужден самим DPC-вызовом.

Это делается с помощью специальной проверки в DPC-процедуре, которая проверяет установку поля ThreadWaitBlockForRelease в структуре данных DFSS, создаваемой для каждого центрального процессора. Если поле установлено, DPC знает, что это запрос на пробуждение, а не на остановку, и выдает событие возобновления, связанное с блоком ожидания квоты (Quota Wait Block). Кроме того, на текущем центральном процессоре принуждается к запуску планировщик простоя (Idle Scheduler), что делается путем установки поля IdleSchedule в блоке KPRCB, который упоминался в ранее изученном разделе, посвященном планировщику простоя.

Но не упомянута еще одна деталь: как только из очереди простоя будет выбран поток и как только будет инициировано переключение контекста, аккумулирование циклов еще раз обнаружит, что поток исчерпал свои циклы, и поток будет снова вставлен в очередь простоя. Поэтому функция освобождения потока из очереди простоя — PsReleaseThreadFromIdleOnlyQueue — должна обновить количество циклов, оставшихся для текущей записи квот центрального процессора, позволяя данному центральному процессору выполнить поток немного дольше. Насколько именно дольше, определяется значением переменной KiCyclesPerClockQuantum, которая была показана ранее в разделе «Кванты времени». Поэтому данному центральному процессору разрешается выполнять текущий поток не более чем на протяжении целого кванта времени.

Кроме того, для этой записи должна быть обновлена базовая норма циклов, поскольку квота для центрального процессора фактически исчерпана и больше не работает на 150 мс кредите цикла. Поэтому норма теперь обновляется, чтобы включить дополнительное значение переменной KiCyclesPerClockQuantum, поделенное на вес сеанса. Поскольку базовая норма цикла изменена, производится новый анализ отсортированного списка блоков, и записи пересортировываются

с целью правильного учета этих изменений. Следовательно, теперь этот блок переместится в начало списка и получит самые высокие шансы быть выбранным в качестве будущего потока из очереди простоя (внутри этого интервала), нуждающегося в том, чтобы его выбрали.

Настройка веса сеанса

До сих пор вес, связанный с сеансом, описывался в виде его исходной величины равной 5. Но этот вес может устанавливаться в диапазоне от 1 и 9, и DFSS предоставляет две внутренние API-функции для управления информацией о весе: PsQueryCpuInformation и ее Set-эквивалент.

Получая в качестве исходных данных массив дескрипторов сеансов (объектов сеансов) и связанных с ними весовых показателей, функция установки (Set API) устанавливает новый вес для каждого сеанса, а также обновляет общий вес, хранящийся в глобальной переменной PspCpuQuotaControl. Путем повторного вызова функции PspCalculateCpuQuotaBlockCycleCredits новые установки получат распространение. По аналогии с этим, функция запроса (Query API) возвращает массив весовых показателей и идентификаторов сеансов. В обоих случаях требуется использование функции SeIncreaseQuotaPrivilege, а также применение константы SESSION_MODIFY_ACCESS для каждого сеанса, чей весовой показатель изменился. Доступ к этим API-функциям осуществляется через исходную API-функцию NtQuerySystemInformation, с вызовом функции SystemCpuQuotaInformation.

Но этот API-интерфейс не предоставляется Windows API напрямую, именно он используется диспетчером системных ресурсов Windows — Windows System Resource Manager, когда администратор назначает разные приоритеты разным пользователям при включенной политике Weighted_Remote_Sessions. Три приоритета — Premium (высший), Standard (стандартный) и Basic (базовый) — отображаются во внутреннем механизме DFSS-планировщика на весовые показатели 1, 5 и 9 соответственно.

Ограничения норм использования центрального процессора

В качестве части аппаратной системы управления квотами в Windows (основанной на исходной программной поддержке ограничения квот, присутствующей со времен первой версии Windows NT) поддержка ограничений на использование центрального процессора имеется в системе в трех разных вариантах: для каждого сеанса, для каждого пользователя и для каждой системы. К сожалению, не существует такого инструментального средства, представляющего собой часть операционной системы, позволяющего вам настраивать эти ограничения, вместо этого приходится самостоятельно изменять настройки реестра.

ПРИМЕЧАНИЕ

Дополнительная документация и примеры, касающиеся того, когда нужно вводить ограничения норм использования центрального процессора, может быть найдена в теме «CPU rate limits in Windows Server 2008 R2 and Windows 7» в статьях Microsoft Technet Knowledge Articles по адресу http://technet.microsoft.com/en-us/library/ff384148(WS.10).aspx.

Новая система квот может быть доступна через раздел реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\QuotaSystem, а также через стандартный системный вызов NtSetInformationProcess. Поэтому ограничение норм использования центрального процессора может быть установлено одним из трех способов:

- Путем создания DWORD-параметра CpuRateLimit и ввода нормирующей информации.
- Путем создания нового раздела с идентификатором безопасности (SID) той учетной записи, на которую нужно наложить ограничения, и создания внутри этого раздела DWORD-параметра CpuRateLimit.
- Путем вызова функции NtSetInformationProcess и передачи ей дескриптора процесса, на который накладываются ограничения, и информации об ограничительной норме использования центрального процессора, если процесс привязан к блоку квот системы.

Во всех трех случаях данные об ограничении норм использования центрального процессора выражаются в виде простого значения; это просто предел нормы использования, выраженный в процентах. Например, чтобы ограничить пользовательские приложения, позволив им потреблять всего 10 % времени центрального процессора, нужно установить для параметра CpuRateLimit значение 10. Диспетчер процессов, отвечающий за исполнение ограничений норм использования центрального процессора, для выполнения этой задачи использует различные системные механизмы. Начнем с того, что ограничения норм работают надежно, поскольку рассмотренные ранее усовершенствования подсчета циклов центрального процессора, позволяющие диспетчеру процессов точно определять, сколько процессорного времени затратил процесс, и знать, должен ли быть принудительно применен лимит. Затем им используется комбинация процедур DPC и APC для снижения частоты использования центрального процессора со стороны DPC и APC, что выходит за рамки возможностей непосредственного управления со стороны разработчиков программ пользовательского режима, но по-прежнему отражается на использовании центрального процессора в системе (в случае ограничений, накладываемых на использование центрального процессора в масштабе всей системы).

И наконец, основной механизм, посредством которого работают ограничения норм, заключается в создании искусственных ожиданий на объекте события (создавая уникальную привязку потока к этому объекту и помещая его в состояние ожидания, на которое не расходуются циклы центрального процессора). Потоки, которые искусственно введены в состояние ожидания из-за ограничений норм использования центрального процессора, могут наблюдаться, поскольку для кода причины их ожидания установлено значение WrCpuRateControl. Этот механизм действует через обычную процедуру поставленного в очередь APC-объекта для потока или потоков внутри процесса, который в данный момент отвечает за эту работу. Со временем событие сигнализирует о себе с помощью процедуры DPC, связанной с таймером (запускаемым каждую половину секунды) и отвечающей за перезарядку запросов на использование центрального процессора в рамках всей системы.

Динамическое добавление и удаление процессоров

Как вы уже видели, разработчики могут тонко отрегулировать вопрос, каким процессам на каком процессоре разрешено выполняться (а в случае идеального процессора предписано выполняться). Все это неплохо работает на системах, имеющих постоянное количество процессоров в ходе всего их рабочего времени. (Например, для внесения каких-либо аппаратных изменений в процессор или в количественный состав процессоров настольный компьютер следует выключить.)

Но современные серверные системы не могут позволить себе простаивать, что обычно требуется для замены центрального процессора или для его добавления. Фактически одним из примеров потребности в добавлении центрального процессора к серверу может послужить период его высокой загруженности, которая превышает возможности машины при заданном текущем уровне производительности. Потребность выключения сервера в период пика его использования извратила бы весь смысл его использования. Чтобы соответствовать требованиям его использования, последнее поколение серверных материнских плат и систем поддерживает добавление процессоров (а также их удаление), не прерывая при этом работы машины. ACPI BIOS и соответствующее оборудование машины было специально создано с учетом предоставления такой возможности, но для полной поддержки необходимо участие операционной системы.

Поддержка динамического изменения количества процессоров предоставляется на уровне HAL, который уведомляет ядро о новом процессоре в системе через функцию KeStartDynamicProcessor. Эта функция выполняет похожую работу, которая проводится, когда система обнаруживает в ходе своего запуска более одного процессора и нуждается в инициализации связанных с этим структур. При динамическом добавлении процессора дополнительная работа проделывается различными компонентами системы. Например, диспетчер памяти выделяет новые страницы и структуры памяти, оптимизированные под центральный процессор. Он также инициализирует находящийся в ядре новый стек DPC, в то время как ядро инициализирует глобальную таблицу дескрипторов — global descriptor table (GDT), таблицу диспетчера прерываний — interrupt Dispatch table (IDT), область управления процессора — processor control region (PCR), блок управления процессора — process control block (PRCB) и другие связанные с процессором структуры.

Также вызываются и остальные части ядра, относящиеся к исполняющей системе, в основном для инициализации создаваемых для каждого процессора списков отвлечений (look-aside list), касающихся добавленного процессора. Например, списки отвлечений, составляемые для каждого процессора, используются диспетчером ввода-вывода, кодом списка отвлечений исполняющей системы, диспетчером кэша и диспетчером объектов для их часто распределяемых структур.

И наконец, ядро инициализирует потоковую поддержку DPC для процессора и корректирует экспортируемые переменные ядра, чтобы в них отображался новый процессор. Обновляются также различные маски диспетчера памяти и исходники процесса, основанные на количествах процессоров. Также нуждаются в обновлении и свойства процессора, чтобы новый процессор соответствовал всей остальной системе (например, на только что добавленном процессоре включается поддержка виртуализации). Инициализационная последователь-

ность завершается уведомлением компонента архитектуры аппаратных ошибок Windows — Windows Hardware Error Architecture (WHEA) — о том, что новый процессор уже в строю.

HAL также привлекается к этому процессу. Он вызывается один раз для запуска динамически добавленного процессора, после того как ядро узнает о нем, и вызывается еще раз после завершения ядром инициализации процессора.

Но эти уведомления и функции обратных вызовов всего лишь ставят ядро в известность и отвечают на изменения процессора. Хотя дополнительный процессор повышает пропускную способность ядра, он не делает ничего для помощи драйверам.

Для обработки драйверов в системе есть новый включенный по умолчанию объект функции обратного вызова исполняющей системы, ProcessorAdd, который может быть зарегистрирован драйверами для получения уведомлений. Как и те функции обратного вызова, которые уведомляют драйверы о состоянии электропитания системы или об изменении системного времени, эта функция обратного вызова позволяет драйверу запрограммировать, к примеру, создание нового рабочего потока, если есть такая необходимость, чтобы он мог справляться с возросшим объемом работы за то же время.

Как только драйверы будут уведомлены, вызывается последний компонент ядра, называемый диспетчером устройств Plug and Play, который добавляет процессор к узлу устройств системы и проводит перебалансировку прерываний, чтобы новый процессор мог обрабатывать прерывания, которые уже были зарегистрированы для других процессоров. Приложения, требующие повышенных ресурсов центрального процессора, также могут получить преимущества от добавления новых процессоров.

Но внезапное изменение родственности может разрушить изменения для выполняемого приложения (особенно при переходе от однопроцессорной к мультипроцессорной среде) из-за появления потенциальных состязательных условий или простого недостаточно рационального распределения работы (процесс мог вычислить наилучшие соотношения при запуске, основанные на количестве центральных процессоров, о которых он был осведомлен). В результате по умолчанию приложения не получат никакой выгоды от динамически добавленного процессора, он должны их запросить.

API-функции Windows SetProcessAffinityUpdateMode и QueryProcessAffinityMode, использующие недокументированный системный вызов NtSet/QueryInformationProcess, сообщают диспетчеру процессов, что этим приложениям нужно обновить свою родственность (флаг AffinityUpdateEnable в структуре EPROCESS) или что им не требуется иметь дело с обновлениями родственности путем установки флага AffinityPermanent в структуре EPROCESS. Как только приложение сообщит системе, что его родственность является постоянной, оно не может чуть позже изменить свое мнение и запросить обновление родственности, следовательно, это одноразовое изменение.

Как часть функции KeStartDynamicProcessor, после перебалансировки прерываний был добавлен новый этап, заключающийся в вызове диспетчера процессов для выполнения обновления родственности с помощью функции PsUpdateActiveProcessAffinity. Некоторые выполняемые в режиме ядра Windows-процессы и службы уже имеют включенное обновление родственности, а вот программное обеспечение сторонних производителей будет нуждаться в перекомпиляции,

чтобы получить выгоду от вызова нового API-интерфейса. Процесс System, процессы Svchost и Smss обладают совместимостью с динамическим добавлением процессоров.

Объекты заданий

Объект задания представляет собой объект ядра, у которого может быть имя, механизм защиты и механизм общего использования и который позволяет контролировать один или несколько процессов, сведенных в группу. Основной функцией объекта задания является предоставление возможности управления группой процессов как единым целым и работы с этим объединением. Процесс может быть членом только одного объекта задания. По умолчанию связь процесса с объектом задания не может быть разорвана, и все созданные им процессы и их потомки также связаны с тем же объектом задания. Объект задания также записывает основную учетную информацию для всех процессов, связанных с заданием, и для всех процессов, которые были связаны с заданием, но работа которых уже была завершена.

Задания могут быть также с помощью Windows-функции GetQueuedCompletionStatus связаны с объектом порта завершения ввода-вывода, в ожидании которого могут находиться другие потоки. Это позволяет заинтересованным сторонам (обычно создателю задания) следить за нарушением лимита и за событиями, которые могут влиять на безопасность задания (например, это может быть созданный новый процесс или процесс, выход из которого произошел аварийно).

Ограничения заданий

На задание могут накладываться следующие ограничения, связанные с центральным процессором и с памятью:

- **Максимальное количество активных процессов.** Ограничивает количество параллельно выполняемых процессов в задании.
- **Распространяемое на все задание ограничение времени работы центрального процессора в пользовательском режиме.** Ограничения максимального количества времени работы центрального процессора в пользовательском режиме, которое может быть потреблено процессами в задании (включая те процессы, которые уже выполнены и из них осуществлен выход).

  По достижении этого лимита все процессы в задании завершаются с кодом ошибки, и никакие новые процессы в задании не могут быть созданы (если только лимит не будет перезапущен). Об объекте задания дается сигнал, поэтому любые потоки, ожидающие задание, будут освобождены. Это исходное поведение может быть изменено с помощью вызова функции SetInformationJobObject для установки класса информации EndOfJobTimeAction и запроса вместо этого отправки уведомления через порт завершения задания.
- **Ограничение времени работы центрального процессора в пользовательском режиме для того или иного процесса.** Позволяет каждому процессу в задании аккумулировать только фиксированное количество времени работы центрального процессора в пользовательском режиме. По достижении максимума процесс завершает свою работу (не имея никаких шансов на окончание незавершенной работы).

- **Родственность процессоров для задания.** Устанавливает маску родственности процессоров для каждого процесса в задании. (Отдельные потоки могут изменять свою родственность на любой поднабор родственности задания, но процессы изменять свои установки родственности процесса не могут.)
- **Групповая родственность для задания.** Устанавливает список групп, которым могут быть назначены процессы в задании. Затем любые изменения родственности сводятся к выбору группы. Это считается устаревшей групповой версией ограничения родственности процессоров для задания, которое не рекомендовано к использованию.
- **Класс приоритета для процессов задания.** Устанавливает класс приоритета для каждого процесса в задании. Потоки не могут повысить свой приоритет относительно класса (обычно они могли это делать). Попытки повышения приоритета потока игнорируются. (Ошибка при вызове функции SetThreadPriority не возвращается, но повышение не происходит.)
- **Исходный минимум и максимум рабочего набора.** Определяет указанный минимум и максимум рабочего набора для каждого процесса в задании. (Эта настройка не распространяется сразу на все задание, у каждого процесса есть свой собственный рабочий набор с одинаковыми минимальными и максимальными значениями.)
- **Лимит виртуальной памяти, передаваемой процессу и заданию.** Определяет максимальный размер виртуального адресного пространства, которое может быть передано либо отдельному процессу, либо всему заданию.

На процесс в задании можно также наложить ограничения безопасности. Задание можно настроить так, чтобы каждый процесс выполнялся под единым для всего задания маркером доступа. Затем можно создать задание, ограждающее процессы от подмены пользователя или от создания процессов, имеющих маркеры доступа, содержащие группу локального администратора. Кроме того, можно применить фильтры безопасности, чтобы в том случае, когда потоки в процессах, содержащихся в задании, олицетворяют клиентские потоки, конкретные привилегии и идентификаторы безопасности — security ID (SID) могли быть исключены из олицетворенного маркера доступа.

И наконец, вы также можете наложить на процесс в задании ограничения, касающиеся пользовательского интерфейса. Такие ограничения включают возможность отстранения процессов от открытия дескрипторов окон, владельцами которых являются потоки за пределами задания, от чтения и (или) записи в буфер обмена, и от изменения многих параметров системы пользовательского интерфейса через Windows-функцию SystemParametersInfo. Эти ограничения, касающиеся пользовательского интерфейса, управляются с помощью драйвера GDI/USER, Win32k.sys, относящегося к подсистеме Windows, и приводятся в исполнение через одну из специальных выносок, зарегистрированных с помощью диспетчера процессов, через выноску задания (job).

Наборы заданий

Реализация задания также позволяет создавать наборы заданий для более тонкого управления тем, с каким объектом задания будет связан отдельно взятый

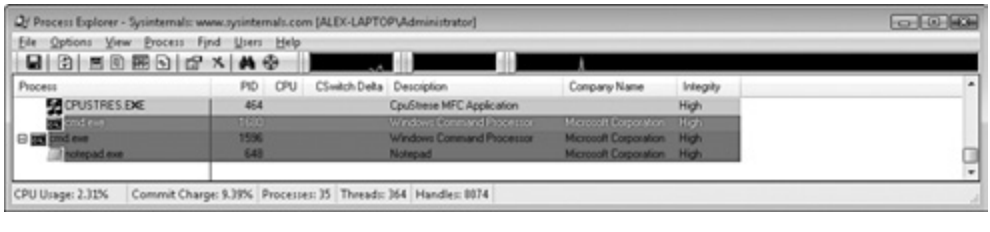
процесс. Набор заданий представляет собой массив, который связывает уровень участника задания с каждым объектом задания, который был создан вызывающей процедурой. Позже, когда диспетчер процесса пытается связать процесс с заданием, он выбирает из набора правильный объект задания на основе уровня участника задания, связанного с только что созданным процессом (этот уровень должен быть выше или равен уровню того участника задания, который породил новый процесс). Это позволяет родительскому процессу создавать несколько объектов заданий, а для своих дочерних процессов подобрать самый подходящий в зависимости от того, какие ограничения требовалось ввести родительскому процессу.

ЭКСПЕРИМЕНТ: ПРОСМОТР ОБЪЕКТА ЗАДАНИЯ

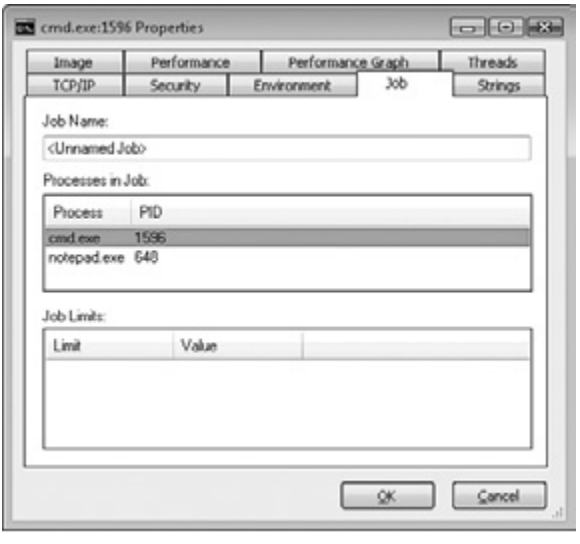
Именованные объекты заданий можно просмотреть с помощью оснастки Производительность (Performance) tool. (Для этого нужно искать объекты производительности Job Object и Job Object Details.) Безымянные задания можно просмотреть с помощью команд отладчика ядра !job или dt nt!_ejob.

Чтобы увидеть, не связан ли процесс с заданием, можно воспользоваться командой отладчика ядра !process или средством Process Explorer. Для создания и просмотра безымянного объекта задания выполните следующие действия:

1. Из окна командной строки воспользуйтесь командой runas для создания процесса, запускающего окно командной строки (Cmd.exe). Например, наберите runas /user:<*домен*>\<*имя_пользователя*> cmd. Команда запросит ваш пароль. Введите свой пароль, и появится окно командной строки. Служба Windows, выполняющая команду runas, создает безымянное задание для содержания всех процессов (чтобы она могла завершить все эти процессы во время выхода из системы).
2. Запустите из окна командной строки программу Блокнот — Notepad.exe.
3. Затем запустите Process Explorer и обратите внимание на то, что процессы Cmd.exe и Notepad.exe выделены в качестве части задания. (Вы можете настроить цвета, используемые для выделения процессов, являющихся участниками задания, щелкнув на пунктах Options (Настройки), Configure Colors (Настроить цвета).) Посмотрите на копию экрана, показывающую эти два процесса.



4. Дважды щелкните либо на процессе Cmd.exe, либо на процессе Notepad.exe, чтобы получить свойства процесса. Вы увидите в диалоговом окне свойств процесса вкладку Job (Задание).
5. Щелкните на вкладке Job (Задание), чтобы просмотреть подробности задания. В данном случае квот, связанных с заданием, не будет, но будут показаны два процесса-участника.



6. Теперь запустите на работающей системе отладчик ядра, выведите список процессов, воспользовавшись командой !process, и найдите недавно созданный процесс, выполняющий Cmd.exe. Затем выведите процесс, воспользовавшись командой !process <*идентификатор процесса*>, найдите адрес объекта задания и в завершение выведите объект задания, воспользовавшись командой !job. Вот как будет выглядеть часть вывода отладчика, полученного с помощью этих команд на работающей системе:

```
lkd> !process 0 1 cmd.exe
PROCESS 8567b758  SessionId: 1  Cid: 0fc4    Peb: 7ffdf000  ParentCid: 00b0
    DirBase: 1b3fb000  ObjectTable: e18dd7d0  HandleCount:  19.
    Image: Cmd.exe
...
    BasePriority                      8
    CommitCharge                      636
...    Job                               85557988

lkd> !job 85557988
Job at 85557988
  TotalPageFaultCount      0
  TotalProcesses           2
  ActiveProcesses          2
  TotalTerminatedProcesses 0
  LimitFlags               0
...
```

7. Чтобы вывести объект задания, можно также воспользоваться командой dt и посмотреть дополнительные поля, относящиеся к заданию, например уровень участия этого задания, если оно является частью набора заданий:

```
lkd> dt nt!_ejob 85557988
```

```
nt!_EJOB
   +0x000 Event            : _KEVENT
...
   +0x0b8 EndOfJobTimeAction : 0
   +0x0bc CompletionPort   : 0x87e3d2e8
   +0x0c0 CompletionKey    : 0x07a89508
   +0x0c4 SessionId        : 1
   +0x0c8 SchedulingClass  : 5
...
   +0x120 MemberLevel      : 0
   +0x124 JobFlags         : 0
```

8. И наконец, если задание имеет UI-ограничения, можно воспользоваться командой dt для вывода структуры задания Win32k (tagW32JOB). Для этого нужно сначала получить указатель на структуру W32PROCESS, как показано в эксперименте в начале этой главы, а затем вывести внутри этой структуры поле pW32Job.

Например, здесь представлена структура задания Win32k для процесса, использующего ограничение блочного доступа к глобальной атомарной таблице пользовательского интерфейса — Block Access To Global Atom Table UI. В структуре, в поле pAtomTable показывается используемая этим процессом локальная атомарная таблица. Вы можете и дальше исследовать эту структуру с помощью команды dt nt!_RTL_ATOM_TABLE и посмотреть, какие атомы в ней определены:

```
lkd> ?? ((win32k!tagPROCESSINFO*)(((nt!_EPROCESS*)0x847c4740)->Win32Process))-
>pW32Job
struct tagW32JOB * 0xfd573300
   +0x000 pNext            : 0xff87c5d8 tagW32JOB
   +0x004 Job              : 0x8356ab90 _EJOB
   +0x008 pAtomTable       : 0x8e03eb18
   +0x00c restrictions     : 0xff
   +0x010 uProcessCount    : 1
   +0x014 uMaxProcesses    : 4
   +0x018 ppiTable         : 0xfe5072c0  -> 0xff97db18 tagPROCESSINFO
   +0x01c ughCrt           : 0
   +0x020 ughMax           : 0
   +0x024 pgh              : (null)
```

■

Заключение

В данной главе была изучена структура процессов, потоков и заданий, рассмотрены способы их создания и прослежены способы принятия системой Windows решений о том, какой поток следует выполнять и насколько долго, а также на каком процессоре или процессорах.

В следующей главе будет рассмотрена та часть системы, которая временами получает больше внимания, чем все остальные части: это имеющийся в Windows монитор безопасности ссылок.

Глава 6. Безопасность

В любой среде, предоставляющей доступ к одним и тем же физическим или сетевым ресурсам сразу нескольким пользователям, остро стоит вопрос предотвращения неавторизованного доступа к конфиденциальным данным. Операционная система, наряду с отдельными пользователями, должна иметь возможность защитить файлы, память и настройки конфигурации от нежелательного просмотра и изменения. Средства безопасности операционной системы включают в себя такие вполне очевидные механизмы, как учетные записи, пароли и защиту файлов. Они также включают в себя такие менее заметные механизмы, как защита операционной системы от повреждения, недопущение осуществления ряда действий (например, перезагрузки компьютера) со стороны менее привилегированных пользователей и запрещение неблагоприятного воздействия пользовательских программ на программы других пользователей или на операционную систему.

В данной главе будет рассмотрено, каким образом каждый аспект конструкции и реализации Microsoft Windows влияет на выполнение жестких требований обеспечения стойкой безопасности.

Оценка безопасности

Наличие программного обеспечения (включая операционные системы), оцениваемого с точки зрения вполне определенных стандартов, помогает правительству, корпорациям и домашним пользователям защитить конфиденциальные и персональные данные, хранящиеся в компьютерных системах. Текущий стандарт оценки безопасности, используемый в Соединенных Штатах и во многих других странах, называется общими критериями (Common Criteria, CC). Но чтобы понять, какие возможности по обеспечению безопасности встроены в Windows, полезно будет узнать историю системы оценки безопасности, повлиявшей на конструкцию Windows, критериях оценки заслуживающих доверия компьютерных систем — Trusted Computer System Evaluation Criteria (TCSEC).

Критерии оценки заслуживающих доверия компьютерных систем

Национальный центр компьютерной безопасности (National Computer Security Center, NCSC) был основан в 1981 году как часть национального агентства безопасности (National Security Agency, NSA) министерства обороны США (U.S. Department of Defense, DoD). Одной из целей NCSC было создание диапазона оценок безопасности, показанного в табл. 6.1, используемого для индикации степени защиты коммерческих операционных систем, сетевых компонентов и предложения заслуживающих доверия приложений. Эти оценки безопасности, которые можно найти по адресу http://csrc.nist.gov/publications/history/dod85.pdf, были определены в 1983 году и обычно упоминаются как «Оранжевая книга».

Стандарт TCSEC состоит из оценочных показателей «уровней доверия», где самые высокие уровни основываются на нижних уровнях и выстраиваются путем добавления более жестких требований защиты и проверок. Ни одна из

операционных систем не отвечает требованиям A1 или оценочному уровню «Verified Design» — проверенной конструкции. Хотя некоторые операционные системы получили одну из оценок B-уровня, вполне достаточным и самым высоким оценочным уровнем практически для всех операционных систем общего назначения является C2.

Таблица 6.1. Оценочные уровни стандарта TCSEC

Оценочный уровень	Описание
A1	Verified Design (проверенная конструкция)
B3	Security Domains (домены безопасности)
B2	Structured Protection (структурированная защита)
B1	Labeled Security Protection (защита с использованием грифа секретности)
C2	Controlled Access Protection (защита управляемого доступа)
C1	Discretionary Access Protection (устаревший уровень) (защита избирательного доступа)
D	Minimal Protection (минимальная защита)

В июле 1995 года Windows NT 3.5 (в версии рабочей станции и в серверной версии) с пакетом обновления Service Pack 3 стала первой версией Windows NT, заслужившей оценочного уровня C2. В марте 1999 года Windows NT 4 с пакетом обновлений Service Pack 3 получила оценочный уровень E3 от организации по безопасности информационных технологий правительства Великобритании, который соответствует принятому в США оценочному уровню C2. В ноябре 1999 года Windows NT 4 с пакетом обновлений Service Pack 6a удостоилась оценочного уровня C2 как в автономной, так и в сетевой конфигурации.

Перечисленные далее требования оценки безопасности C2 считались основными, и они по-прежнему считаются основными требованиями для любой защищенной операционной системы:

- Средство безопасного входа в систему, которое требует уникальной идентификации пользователей и получения ими полномочий доступа к компьютеру только после того, как они тем или иным способом пройдут аутентификацию.
- Управление избирательным доступом, позволяющее владельцу ресурса (например, файла) определить, кто может получить доступ к ресурсу и что он при этом может с ним сделать. Владелец наделяет правами, разрешающими различные виды доступа, отдельного пользователя или группу пользователей.
- Контроль безопасности, позволяющий обнаруживать и записывать события, относящиеся к вопросам безопасности, или любые попытки создания системных ресурсов, а также обращения к ним или их удаления. Запись идентификаторов при входе в систему, позволяющая устанавливать идентичность всех пользователей, упрощая тем самым отслеживание любого пользователя, пытающегося выполнить неавторизованное действие.
- Защита от повторного использования объекта, которая не позволяет пользователям просматривать данные, удаленные другим пользователем, или не позволяет обращаться к памяти, которая ранее была использована, а затем освобождена другим пользователем. Например, в некоторых операционных

системах позволяется создавать новый файл определенной длины, а затем проверять содержимое файла, чтобы увидеть данные, находящиеся на том месте диска, которое было выделено файлу. Эти данные могут представлять собой ценную информацию, которая была сохранена в файле другого пользователя, но затем была удалена. Защита от повторного использования объекта закрывает эту потенциальную дыру безопасности путем инициализации всех объектов, включая файлы и память перед их выделением пользователю.

Windows также отвечает двум требованиям безопасности B-уровня:

- Механизм доверенного маршрута, который не дает троянским программам возможности перехвата пользовательских имен и паролей при попытке их входа в систему. Используемый в Windows механизм доверенного маршрута появился в форме последовательности переноса внимания на вход в систему Ctrl+Alt+Delete, которая не может быть перехвачена непривилегированными приложениями. Эта клавиатурная последовательность, известная также как последовательность переноса внимания на безопасность работы — secure attention sequence (SAS), всегда выводит управляемый системой экран безопасности Windows (если пользователь уже вошел в систему) или экран входа в систему, чтобы можно было легко распознать троянские программы. (Если групповая политика это позволяет, последовательность переноса внимания на безопасность работы может быть также отправлена программным способом путем использования API-функции SendSAS.) При вводе SAS троянская программа, предоставляющая поддельное диалоговое окно, будет обойдена.
- Доверенное средство управления, которое требует поддержки ролей отдельных учетных записей для осуществления административных функций. Например, для администрирования (администраторов), для учетных записей пользователей, отвечающих за резервное копирование компьютера, и для обычных пользователей предоставляются отдельные учетные записи.

Windows отвечает всем этим требованиям благодаря своей подсистеме безопасности и связанным с ней компонентам.

Общие критерии

В январе 1996 года США, Великобритания, Германия, Франция, Канада и Нидерланды опубликовали совместно разработанную спецификацию оценки безопасности под названием «Общие критерии оценки безопасности информационных систем» — Common Criteria for Information Technology Security Evaluation (CCITSE). Спецификация CCITSE, которую обычно называют просто общими критериями — Common Criteria (CC), — является общепризнанным международным стандартом для оценки безопасности программных продуктов. Главная страница CC находится по адресу www.niap-ccevs.org/cc-scheme/.

Стандарт CC обладает большей гибкостью по сравнению с оценками доверительности TCSEC и имеет структуру, более близкую к стандарту ITSEC, чем к стандарту TCSEC. Стандарт CC включает понятие профиля защиты — Protection Profile (PP), — используемое для сбора требований безопасности в свободно определяемые и сопоставляемые наборы, и понятие задания по безопасности — Security Target (ST), — содержащего набор требований по безопасности, которые могут быть выдвинуты на основе PP. Стандарт CC также определяет диапазон из семи оценочных уровней доверия — Evaluation Assurance

Levels (EAL), — которые показывают уровень доверия к сертификации. Таким образом, CC (подобно предшествующему ему стандарту ITSEC) удаляет связь между функциональностью и уровнем гарантий, которая была представлена в TCSEC и в ранних схемах сертификации.

Операционные системы Windows 2000, Windows XP, Windows Server 2003 и Windows Vista Enterprise добились сертификации Common Criteria под профилем защиты контролируемого доступа — Controlled Access Protection Profile (CAPP). Это примерно соответствует оценочному уровню TCSEC C2. Все они получили оценку EAL 4+, где плюс означает «исправление недостатков». EAL 4 является высшим уровнем, признаваемым за пределами национальных границ.

В марте 2011 года операционные системы Windows 7 и Windows Server 2008 R2 получили оценку соответствия требованиям к профилю защиты, установленным правительством США для операционных систем общего назначения в сетевой среде[1]. Сертификация включает гипервизор Hyper-V, и, опять же, Windows добилась оценочного уровня доверия EAL-4+. Отчет об оценке можно найти на веб-сайте http://www.commoncriteriaportal.org/files/epfiles/st_vid10390-vr.pdf, а описание задания по безопасности, дающее подробные сведения об удовлетворенных требованиях, можно найти на веб-сайте http://www.commoncriteriaportal.org/files/epfiles/st_vid10390-st.pdf.

Системные компоненты безопасности

В реализацию системы безопасности Windows входят следующие компоненты и базы данных:

- **Монитор безопасности — Security reference monitor (SRM).** Компонент, который находится в исполняющей системе Windows (%SystemRoot%\System32\Ntoskrnl.exe) и отвечает за определение структуры данных маркеров доступа для представления контекста безопасности, выполнение проверки безопасности доступа к объектам, работу с привилегиями (правами пользователей) и генерацию любых итоговых сообщений проверки безопасности.
- **Подсистема проверки подлинности локальной системы безопасности — Local Security Authority subsystem (LSASS).** Процесс пользовательского режима запускает образ %SystemRoot%\System32\Lsass.exe, который отвечает за политику безопасности локальной системы (которая, например, определяет пользователей, обладающих правом входить в систему, политики паролей, привилегии, предоставленные пользователям и группам, и настройки проверки безопасности системы), аутентификацию пользователя и отправку сообщений проверки безопасности журналу событий (Event Log). Основная часть этих функциональных возможностей реализована в загружаемой LSASS библиотеке службы авторизации локальных пользователей — The Local Security Authority service (Lsasrv — %SystemRoot%\System32\Lsasrv.dll).
- **База данных политики LSASS.** База данных, в которой содержатся настройки политики безопасности локальной системы. Эта база данных хранится в ре-

[1] US Government Protection Profile for General-Purpose Operating Systems in a Networked Environment, версии 1.0 от 30 августа 2010 года (GPOSPP) (http://www.commoncriteriaportal.org/files/ppfiles/pp_gpospp_v1.0.pdf).

естре в ACL-защищенной области в разделе HKLM\SECURITY. Она включает информацию, которая, в частности, определяет, каким доменам доверена аутентификация попыток входа в систему, у кого есть права на доступ к системе и каким образом осуществляется этот доступ (интерактивные, сетевые или служебные регистрации), кому и какие привилегии назначены и какие виды проверки безопасности следует выполнять. В базе данных политики LSASS также хранятся «секреты», включающие информацию о входе в систему, использующуюся для кэшированных входов в домены и входов в службы Windows под учетной записью пользователя. Дополнительные сведения о службах Windows даны в главе 4 «Механизмы управления».

- **Администратор учетных данных в системе защиты — Security Accounts Manager (SAM).** Служба, отвечающая за управление базой данных, содержащей имена пользователей и определения групп на локальной машине. SAM-служба, реализованная в виде библиотеки %SystemRoot%\System32\Samsrv.dll, загружается в процесс LSASS.
- **База данных SAM.** База данных, содержащая определения локальных пользователей и групп, наряду с их паролями и другими атрибутами. На контроллерах доменов служба SAM не хранит сведений о пользователях, определенных на домене, но хранит информацию определения учетной записи и пароля системного администратора, имеющего право на восстановление системы. Эта база данных хранится в реестре в разделе HKLM\SAM.
- **Active Directory.** Служба каталогов, которая содержит базу данных, хранящую информацию об объектах в домене. *Домен* является совокупностью компьютеров и связанных с ними группами безопасности, которые управляются как единое целое. Active Directory сохраняет информацию об объектах в домене, куда включаются пользователи, группы и компьютеры. В Active Directory хранится информация о паролях и привилегиях для пользователей и групп домена, которая тиражируется на компьютерах, назначаемых в качестве контроллеров домена. Сервер Active Directory, реализованный в виде библиотеки %SystemRoot%\System32\Ntdsa.dll, запускается в контексте процесса LSASS. Дополнительные сведения об Active Directory даны в главе 7 «Сеть».
- **Пакеты аутентификации (Authentication packages).** Включают динамически подключаемые библиотеки — dynamic-link libraries (DLL), которые запускаются как в контексте процесса LSASS, так и в контекстах клиентских процессов и реализуют имеющуюся в Windows политику аутентификации. DLL-библиотека аутентификации отвечает за аутентификацию пользователей, реализуя ее путем проверки соответствия введенного имени пользователя и пароля и возвращения процессу LSASS в случае такого соответствия информации, уточняющей детали пользовательской идентичности с точки зрения безопасности, которые используются LSASS для генерации маркера доступа.
- **Интерактивный диспетчер входа в систему (Winlogon).** Процесс пользовательского режима, в котором выполняется образ %SystemRoot%\System32\Winlogon.exe, отвечающий за реагирование на SAS и за управление интерактивными сеансами входа в систему. К примеру, Winlogon создает первый пользовательский процесс, когда пользователь входит в систему.
- **Пользовательский интерфейс входа в систему — Logon user interface (LogonUI).** Процесс пользовательского режима, выполняющий образ %System-

Root%\System32\LogonUI.exe, который предоставляет пользователям интерфейс, который может быть использован ими для самоидентификации на системе. LogonUI использует поставщиков учетных данных для запроса учетных данных пользователя с помощью различных методов.

- **Поставщики учетных данных — Credential providers (CP).** Находящиеся в процессе COM-объекты, запускающие процесс LogonUI (стартующий по запросу со стороны Winlogon при выполнении SAS) и используемые для получения имени пользователя и пароля, PIN-кода смарт-карты или биометрических данных (например, отпечатка пальца). Стандартными CP являются библиотеки %SystemRoot%\System32\authui.dll и %SystemRoot%\System32\SmartcardCredentialProvider.dll.
- **Служба входа в сеть — Network logon service (Netlogon).** Служба Windows (%SystemRoot%\System32\Netlogon.dll) устанавливает безопасный канал к контроллеру домена, по которому отправляются запросы безопасности, например интерактивный вход в систему (если на контроллере домена запущена Windows NT 4) или проверка аутентификации диспетчера LAN Manager и диспетчера NT LAN Manager (v1 и v2). Netlogon также используется для входов в систему со стороны Active Directory.
- **Драйвер устройства безопасности ядра — Kernel Security Device Driver (KSecDD).** Библиотека функций режима ядра, реализующая интерфейсы усовершенствованной системы вызова локальных процедур — advanced local procedure call (ALPC), — которые используются для обмена данными в пользовательском режиме с LSASS другими компонентами безопасности режима ядра, включая такой компонент, как шифрующая файловая система — Encrypting File System (EFS). KSecDD находится в файле %SystemRoot%\System32\Drivers\Ksecdd.sys.
- **AppLocker.** Механизм, позволяющий администраторам определять, какие исполняемые файлы, DLL-библиотеки и сценарии могут использоваться конкретными пользователями и группами. AppLocker состоит из драйвера (%SystemRoot%\System32\Drivers\AppId.sys) и службы (%SystemRoot%\System32\AppIdSvc.dll), выполняемой в процессе SvcHost.

На рис. 6.1 показаны взаимоотношения между некоторыми из этих компонентов и баз данных, которыми они управляют.

Монитор безопасности SRM, выполняющийся в режиме ядра, и процесс LSASS, выполняющийся в пользовательском режиме, обмениваются данными с помощью средства ALPC, рассмотренного в главе 3 «Системные механизмы». В ходе инициализации системы SRM создает порт SeRmCommandPort, к которому подключается LSASS. Когда запускается процесс LSASS, он создает ALPC-порт SeLsaCommandPort. SRM подключается к этому порту, приводя к созданию закрытых коммуникационных портов. SRM создает общий раздел памяти для сообщений, превышающих размер 256 байт, передавая дескриптор в запросе на подключение. После того как SRM и LSASS подключатся друг к другу в ходе инициализации системы, больше они соответствующие коммуникационные порты не прослушивают. Поэтому более поздний пользовательский процесс не имеет возможности успешно подключиться к любому из этих портов для нанесения какого-либо вреда, поскольку запрос на подключение никогда не будет завершен.

На рис. 6.2 показаны пути коммуникации, имеющиеся после инициализации системы.

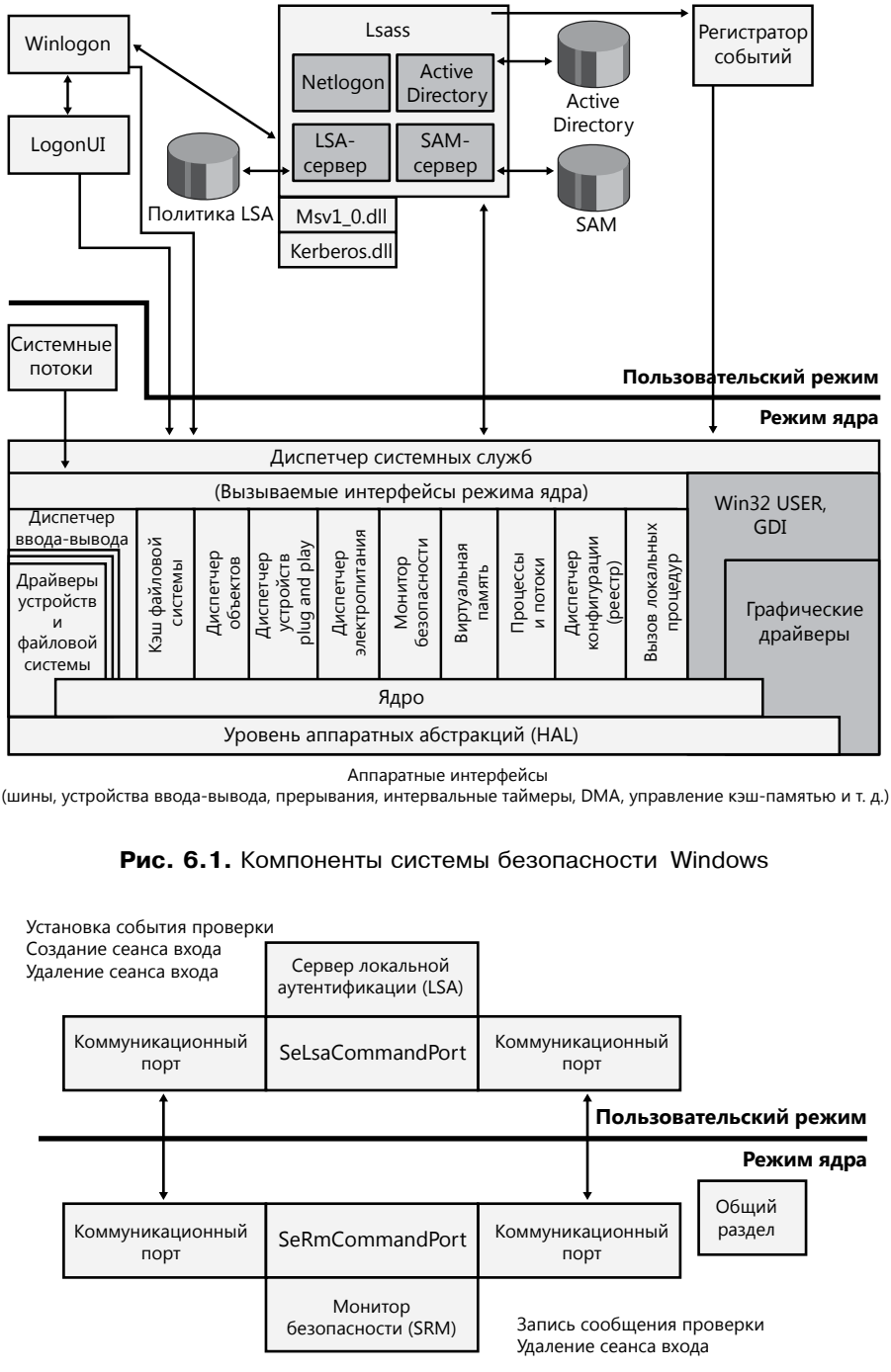


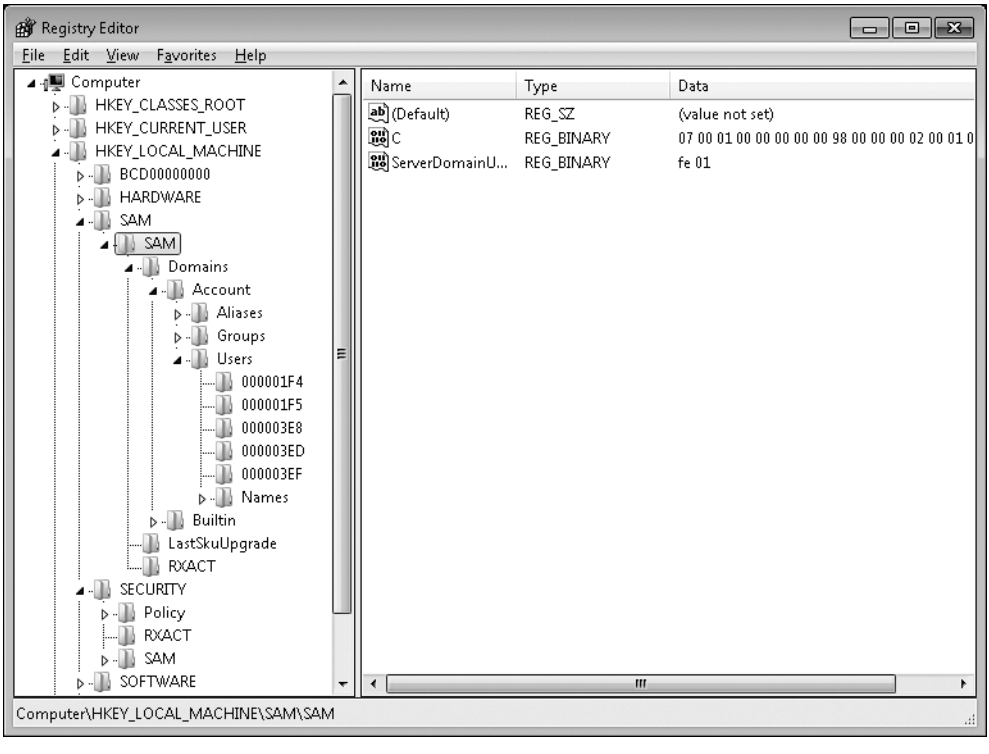
Рис. 6.1. Компоненты системы безопасности Windows

Рис. 6.2. Связь между SRM и LSASS

ЭКСПЕРИМЕНТ: ИЗУЧЕНИЕ HKLM\SAM И HKLM\SECURITY ИЗНУТРИ

Дескрипторы безопасности, связанные с разделами реестра SAM Security, предотвращают доступ к ним под любой учетной записью, отличной от учетной записи локальной системы. Один из способов получения доступа к этим разделам для их исследования заключается в сбросе их защиты, но это может ослабить безопасность системы. Другим способом является выполнение Regedit.exe под учетной записью локальной системы. Это можно сделать путем использования средства PsExec, входящего в набор Windows Sysinternals, с ключом –s:

```
C:\>psexec -s -i -d c:\windows\regedit.exe
```

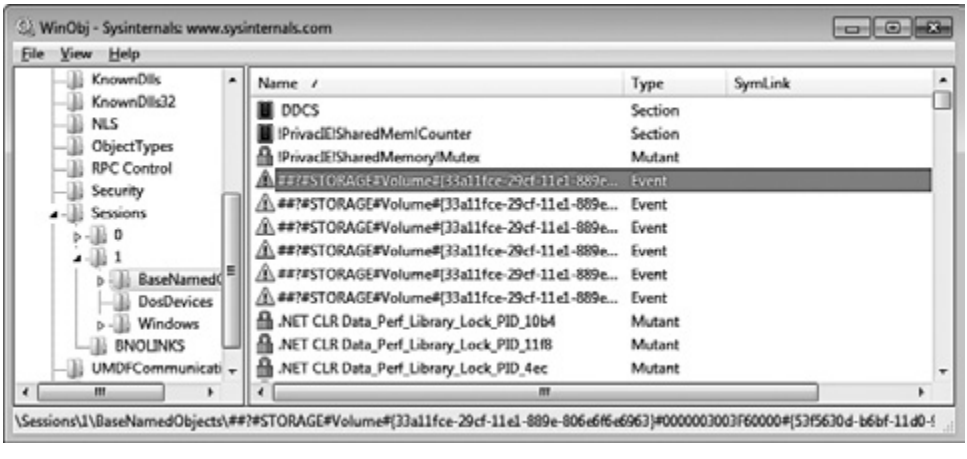


Защита объектов

Суть управления избирательным доступом и проверки безопасности заключается в защите объектов и в регистрации доступа. В число защищаемых в Windows объектов входят файлы, устройства, почтовые ящики, каналы (именованные и безымянные), задания, процессы, потоки, события, ключевые события, пары событий, мьютексы, семафоры, общие разделы памяти, порты завершения ввода-вывода, LPC-порты, таймеры ожиданий, маркеры доступа, тома, станции окон, рабочие столы, сетевые ресурсы, службы, разделы реестра, принтеры, объекты Active Directory и т. д. — теоретически все, что управляется диспетчером объектов исполняющей системы. На практике те объекты, которые не открыты пользовательскому режиму (например, объекты драйверов), обычно не защищены. Код

режима ядра считается надежным и обычно использует интерфейсы к диспетчеру объектов, не выполняющие проверки доступа. Поскольку системные ресурсы, открытые пользовательскому режиму (и поэтому требующие проверки безопасности), реализованы как объекты в режиме ядра, диспетчер объектов Windows играет ключевую роль в обеспечении безопасности объектов.

Диспетчер объектов был рассмотрен в главе 3, где было показано, как он поддерживает для объектов дескриптор безопасности. На рис. 6.3 с помощью средства Sysinternals Winobj показан дескриптор безопасности для объекта раздела в сеансе пользователя. Хотя наиболее часто с защитой объектов ассоциируются такие ресурсы, как файлы, Windows использует аналогичную модель и такой же механизм безопасности, как и для файлов в файловой системе и для объектов исполняющей системы. Что же касается вопросов управления доступом, то объекты исполняющей системы отличаются от файлов только в методах доступа, поддерживаемых каждым типом объектов.



Как вы увидите далее, то, что показано на рис. 6.3, на самом деле является принадлежащим объекту списком управления избирательным доступом — discretionary access control list, или DACL. Списки DACL будут подробно рассмотрены в следующем разделе.

Чтобы контролировать возможность работы с объектом, система безопасности должна сначала убедиться в идентичности каждого пользователя. Такая необходимость гарантии пользовательской идентичности является причиной того, что Windows требует аутентифицированного входа в систему перед получением доступа к любым системным ресурсам. Когда процесс запрашивает дескриптор объекта, диспетчер объектов и система безопасности используют идентификатор безопасности запрашивающего и дескриптор безопасности объекта, чтобы определить, может ли запрашивающему быть присвоен дескриптор, предоставляющий процессу желаемый доступ к объекту.

Как станет ясно при дальнейшем рассмотрении вопроса, поток может получить другой контекст безопасности, нежели его процесс. Этот механизм называется заимствованием прав (impersonation), и когда поток заимствует чьи-либо права, механизм проверки безопасности использует контекст безопасности потока, а не того процесса, которому принадлежит поток. Если поток ничьих прав

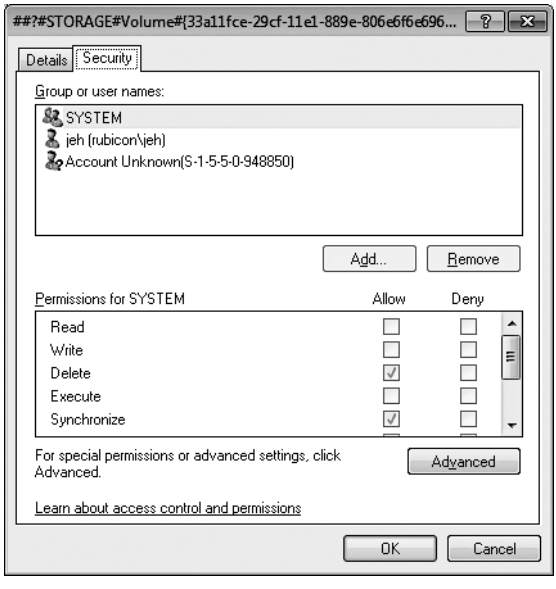


Рис. 6.3. Объект исполняющей системы и его дескриптор безопасности, просматриваемый с помощью средства Winobj

не заимствует, проверка безопасности возвращается обратно к использованию контекста безопасности того процесса, который владеет потоком. Важно иметь в виду, что все потоки в процессе используют общую таблицу дескрипторов, поэтому, когда поток открывает объект, даже если он заимствует чьи-либо права, доступ к объекту имеют все потоки процесса.

Иногда проверки идентичности пользователя недостаточно системе для предоставления доступа к ресурсу, который должен быть доступен согласно учетной записи. Рассуждая логически, можно считать, что есть четкое различие между службой, выполняемой под учетной записью Alice и неизвестным приложением, загруженным пользователем Alice при пользовании Интернетом. Windows достигает подобной изоляции в рамках работы одного и того же пользователя с помощью механизма целостности Windows (Windows integrity mechanism), в котором реализуются уровни целостности. Механизм целостности Windows используется с повышениями привилегий, реализуемыми системой управления учетными записями пользователей — User Account Control (UAC), — в защищенном режиме Internet Explorer — Protected Mode Internet Explorer (PMIE) — и в изоляции привилегий пользовательского интерфейса — User Interface Privilege Isolation (UIPI).

Проверки прав доступа

Модель безопасности Windows требует, чтобы поток, прежде чем открыть объект, указал, какого типа действия он хочет совершить над объектом. Для выполнения проверок прав доступа, основанных на желаемом потоком доступе, диспетчер объектов вызывает SRM, и если доступ предоставляется, процессу потока назначается дескриптор, с которым поток (или другие потоки процесса) может выполнять дальнейшие операции над объектом. В главе 3 уже объяснялось, что

диспетчер объектов записывает предоставленные права доступа в дескриптор в таблице дескрипторов процесса.

Одним из событий, заставляющих диспетчер объектов выполнять проверку безопасности доступа, является открытие процессом существующего объекта с использованием его имени. Когда объект открывается по имени, диспетчер объектов выполняет поиск указанного объекта в пространстве имен диспетчера объектов. Если объект не находится во вторичном пространстве имен, например в пространстве имен диспетчера конфигурации реестра или в пространстве имен драйверов файловой системы, диспетчер объектов, как только он обнаружит объект, вызывает внутреннюю функцию ObpCreateHandle. Как следует из имени данной функции, ObpCreateHandle создает запись в таблице дескрипторов процесса, которая становится связанной с объектом. Сначала функция ObpCreateHandle вызывает функцию ObpGrantAccess, чтобы посмотреть, имеет ли поток разрешение на доступ к объекту, если он такое разрешение имеет, функция ObpCreateHandle вызывает функцию исполняющей системы ExCreateHandle для создания записи в таблице дескрипторов процесса. Для инициализации проверки безопасности доступа функция ObpGrantAccess вызывает функцию ObCheckObjectAccess.

Функция ObpGrantAccess передает функции ObCheckObjectAccess мандат безопасности потока, открывающего объект, типы доступа к объекту, запрошенному потоком (чтение, запись, удаление и т. д.), и указатель на объект. Сначала функция ObCheckObjectAccess блокирует дескриптор безопасности объекта и контекст безопасности потока. Блокировка дескриптора безопасности объекта не позволяет другим потокам системы изменять параметры безопасности объекта в ходе проверки прав доступа. Блокировка контекста безопасности потока не позволяет другим потокам (из этого же процесса или из других процессов) изменять идентификационные данные о безопасности потока в ходе проверки прав доступа. Затем функция ObCheckObjectAccess вызывает метод безопасности объекта, чтобы получить настройки безопасности этого объекта. Вызов метода безопасности может задействовать функцию в других компонентах исполняющей системы. Тем не менее многие объекты исполняющей системы полагаются на исходную поддержку диспетчера безопасности системы.

Когда компонент исполняющей системы, определяющий объект, не хочет переписывать исходную политику безопасности SRM, он делает пометку, что тип объекта имеет исходную настройку безопасности. Когда SRM вызывает метод безопасности объекта, то сначала он проверяет, не установлены ли для объекта исходные настройки безопасности. Объект с исходными настройками безопасности сохраняет информацию о своей безопасности в своем заголовке, и его методом безопасности является SeDefaultObjectMethod. Объект, который не зависит от исходных настроек безопасности, должен управлять своей собственной информацией о безопасности и предоставлять особый метод безопасности. К объектам, зависящим от исходных настроек безопасности, относятся мьютексы, события и семафоры. В качестве примера объекта, переписывающего исходные настройки безопасности, можно привести файловый объект. Диспетчер ввода-вывода, который определяет тип файлового объекта, имеет драйвер файловой системы, в котором находится файл, управляющий безопасностью его файлов (или выбирающий отказ от этого управления). Таким образом, когда

система запрашивает безопасность файлового объекта, представляющего файл на томе NTFS, метод безопасности файлового объекта диспетчера вывода-вывода извлекает информацию о настройках безопасности файла, используя драйвер файловой системы. Но следует заметить, что функция ObCheckObjectAccess не выполняется, когда файлы открыты, поскольку они находятся во вторичном пространстве имен. Система вызывает метод безопасности файлового объекта только тогда, когда поток явным образом запрашивает информацию о безопасности файла или устанавливает для него настройки безопасности (например, с помощью Windows-функции SetFileSecurity или GetFileSecurity).

После получения информации о безопасности объекта функция ObCheckObjectAccess вызывает SRM-функцию SeAccessCheck. Функция SeAccessCheck относится к функциям, составляющим основу модели безопасности Windows. Среди прочих входных параметров функция SeAccessCheck принимает информацию о безопасности объекта, идентификационные данные о безопасности потока в том виде, в котором они извлечены функцией ObCheckObjectAccess, и запрашиваемый потоком тип доступа. Функция SeAccessCheck возвращает значение True или False, в зависимости от того, предоставлен или нет потоку доступ к запрашиваемому объекту.

Еще одним событием, заставляющим диспетчер объектов проводить проверку прав доступа, является ссылка процесса на объект с использованием существующего дескриптора. Такие ссылки зачастую проводятся опосредованно, например, когда процесс вызывает API-функцию Windows для работы с объектом и передает ей дескриптор объекта. Например, поток, открывающий файл, может запросить право на чтение файла. Если у потока есть право на такой доступ к объекту, как предписано его контекстом безопасности и настройками безопасности файла, диспетчер объектов создает дескриптор, который представляет файл в таблице дескрипторов того процесса, которому принадлежит поток. Типы доступа предоставляются процессу через дескриптор, который хранится с дескриптором диспетчера объектов.

Впоследствии поток может попытаться провести запись в файл, используя Windows-функцию WriteFile, передавая ей в качестве параметра дескриптор файла. Системная служба NtWriteFile, которую функция WriteFile вызывает через библиотеку Ntdll.dll, использует функцию диспетчера объектов ObReferenceObjectByHandle, чтобы получить из дескриптора указатель на объект файла. Функция ObReferenceObjectByHandle получает в качестве параметра информацию о доступе, который хочет получить к объекту вызывающий процесс. После поиска записи дескриптора в таблице дескрипторов процесса функция ObReferenceObjectByHandle сравнивает запрошенный доступ с доступом, предоставленным во время открытия файла. В данном примере функция ObReferenceObjectByHandle покажет, что операция записи должна быть отклонена, поскольку вызывающий процесс не получил доступ по записи при открытии файла.

Функции безопасности Windows также позволяют Windows-приложениям определять свои собственные закрытые объекты и вызывать службы SRM (через API-функции пользовательского режима AuthZ, которые будут рассмотрены чуть позже), чтобы задействовать модель безопасности Windows в отношении таких объектов. Многие функции режима ядра, которые диспетчер объектов и другие компоненты исполняющей системы используют для защиты своих собственных

объектов, экспортируются в виде API-функций Windows, работающих в пользовательском режиме. Аналогом функции SeAccessCheck в пользовательском режиме является AuthZ API-функция AccessCheck. Таким образом, Windows-приложения могут воспользоваться гибкостью модели безопасности и явным образом интегрироваться с имеющимися в Windows интерфейсами аутентификации и администрирования.

Сутью модели безопасности SRM является уравнение, имеющее три входных параметра: идентификационные данные безопасности потока, доступ, который поток желает получить к объекту, и настройки безопасности объекта. На выходе получаем либо «да», либо «нет», что показывает, дает модель безопасности потоку запрошенный доступ или не дает. В следующих разделах входные параметры рассматриваются более подробно, после чего дается описание алгоритма проверки доступа, реализованного в модели.

Идентификаторы безопасности

Вместо использования имен (которые могут быть уникальными, а могут и не быть), для идентификации всего, что производит в системе действия, Windows использует идентификаторы безопасности — security identifiers (SID). SID-идентификаторы имеются у пользователей, а также у локальных и доменных групп, у локальных компьютеров, доменов, участников доменных групп и служб. SID представляет собой числовое значение переменной длины, состоящее из номера версии SID-структуры, 48-разрядное значение идентификатора полномочий и переменное количество 32-разрядных кодов значений подчиненных полномочий или относительных идентификаторов — relative identifier (RID). Значение полномочий идентифицирует агента, выдавшего SID, и этим агентом обычно является локальная система Windows или домен. Значения подчиненных полномочий идентифицируют представителей, имеющих отношение к выдавшему полномочия, а RID-идентификаторы являются просто способом, применяемым в Windows для создания уникальных SID на основе общего базового SID. Из-за большой длины SID-идентификаторов Windows старается сгенерировать внутри каждого SID по-настоящему случайное значение, практически невозможно, чтобы Windows выдала один и тот же SID на машине или домене или где-либо еще дважды.

При текстуальном отображении каждый SID содержит префикс S, и его различные компоненты отделены друг от друга дефисами:

```
S-1-5-21-1463437245-1224812800-863842198-1128
```

В данном SID номером версии служит цифра 1, значением идентификатора полномочий служит цифра 5 (полномочия безопасности Windows), а затем следуют четыре значения подчиненных полномочий плюс один RID (1128), который составляет оставшуюся часть SID. Это SID домена, а у локального компьютера домена будет SID с тем же номером версии, значением идентификатора полномочий и количеством значений подчиненных полномочий.

При установке Windows программа Windows Setup выдает компьютеру SID машины. Windows назначает SID-идентификаторы локальным учетным записям, имеющимся на компьютере. Каждый SID локальной учетной записи создается на основе исходного компьютерного SID и в конце имеет RID. RID-идентификатор для пользовательских учетных записей и групп начинается с 1000 и становится

больше на 1 с каждым новым пользователем или группой. По аналогии с этим Dcpromo.exe (Domain Controller Promote), утилита, используемая для создания нового домена Windows, повторно использует SID компьютера, повышаемого до контроллера домена в качестве SID домена, и она заново создает SID для компьютера, если он когда-либо будет понижен в должности. Windows выдает для нового домена SID-идентификаторы учетных записей, основанные на SID-идентификаторе домена с добавлением RID-идентификатора (который снова начинается с 1000 и повышается на 1 для каждого нового пользователя или группы). RID со значением 1028 показывает, что SID является двадцать девятым выпущенным в домене.

Windows выдает SID-идентификаторы, которые состоят из SID компьютера или домена с предопределенным RID-идентификатором для множества предопределенных учетных записей и групп. Например, RID для учетной записи администратора имеет значение 500, а RID для гостевой учетной записи имеет значение 501. Например, учетная запись локального администратора имеет в своей основе SID компьютера с добавленным к нему RID, который имеет значение 500:

```
S-1-5-21-13124455-12541255-61235125-500
```

Windows также определяет ряд встроенных локальных и доменных SID для представления широко известных групп. Например, SID, идентифицирующий любые учетные записи (за исключение анонимных пользователей), является всеобщим — Everyone SID: S-1-1-0. Другим примером группы, которая может быть представлена SID-идентификатором, является сетевая группа, то есть группа, представляющая пользователей, зарегистрировавшихся на машине по сети. SID сетевой группы имеет значение S-1-5-2. В представленной ниже табл. 6.2 из документации по Windows SDK показываются некоторые основные, широко известные SID-идентификаторы, их числовые значения и использование. В отличие от пользовательских SID эти SID-идентификаторы являются предопределенными константами и имеют одинаковые значения на каждой системе Windows и домене во всем мире. Таким образом, файл, доступный членам группы Everyone на той системе, где он был создан, также будет доступен группе Everyone на любой другой системе или домене, на которую будет перемещен жесткий диск, на котором он размещается. Разумеется, пользователи на таких системах должны пройти аутентификацию учетной записи на этих системах, перед тем как стать членами группы Everyone.

ПРИМЕЧАНИЕ

Список предопределенных SID-идентификаторов можно увидеть в базе знаний Microsoft Knowledge Base в статье 243330, которая находится по адресу http://support.microsoft.com/kb/243330.

И наконец, Winlogon создает уникальный SID входа в систему для каждого интерактивного сеанса входа. Обычно SID входа в систему применяется в элементе управления доступом — access control entry (ACE), который разрешает доступ во время сеанса входа клиента в систему. Например, служба Windows для запуска нового сеанса входа в систему может использовать функцию LogonUser. Эта функция возвращает маркер доступа, из которого служба может извлечь SID входа в систему. Затем эта служба может использовать SID в ACE, позволяющем

сеансу входа клиента в систему получать доступ к интерактивной станции окна и к рабочему столу. SID для сеанса входа в систему имеет значение S-1-5-5-0, RID генерируется случайным образом.

Таблица 6.2. Ряд широко известных SID-идентификаторов

SID	Группа	Использование
S-1-0-0	Nobody (Никто)	Используется, когда SID неизвестен
S-1-1-0	Everyone (Все)	Группа, включающая всех пользователей за исключением анонимных
S-1-2-0	Local (Локальная)	Пользователи, вошедшие в терминалы, которые локально (физически) подключены к системе
S-1-3-0	Creator Owner ID (ID владельца создателя)	Идентификатор безопасности, который будет заменен идентификатором безопасности того пользователя, который создал новый объект. Этот SID используется в наследуемых ACE-элементах
S-1-3-1	Creator Group ID (ID группы создателя)	Идентификатор безопасности, который будет заменен идентификатором безопасности основной группы того пользователя, который создал новый объект. Этот SID используется в наследуемых ACE-элементах
S-1-9-0	Resource Manager (Диспетчер ресурсов)	Используется приложениями, созданными сторонними разработчиками, в которых используется своя собственная защита внутренних данных (например, Microsoft Exchange)

ЭКСПЕРИМЕНТ: ИСПОЛЬЗОВАНИЕ PSGETSID И PROCESS EXPLORER ДЛЯ ПРОСМОТРА SID-ИДЕНТИФИКАТОРОВ

SID-представление любой используемой вами учетной записи можно запросто посмотреть, запустив утилиту PsGetSid из набора Sysinternals.

Параметры PsGetSid позволяют переводить имена машин и пользовательских учетных записей в соответствующие им SID-идентификаторы, и наоборот.

Если запустить PsGetSid без параметров, утилита выводит SID, назначенный локальному компьютеру. Пользуясь тем, что у учетной записи Administrator значение RID всегда равно 500, вы можете определить имя, назначенное учетной записи (в тех случаях, когда системный администратор переименовал учетную запись из соображений безопасности), просто передав в качестве аргумента командной строки для запуска утилиты PsGetSid SID машины, к которому добавлена строка –500.

Для получения SID учетной записи домена введите имя пользователя, поставив в качестве префикса имя домена:

```
c:\>psgetsid redmond\daryl
```

SID домена можно определить, указав имя домена в качестве аргумента при запуске утилиты PsGetSid:

```
c:\>psgetsid Redmond
```

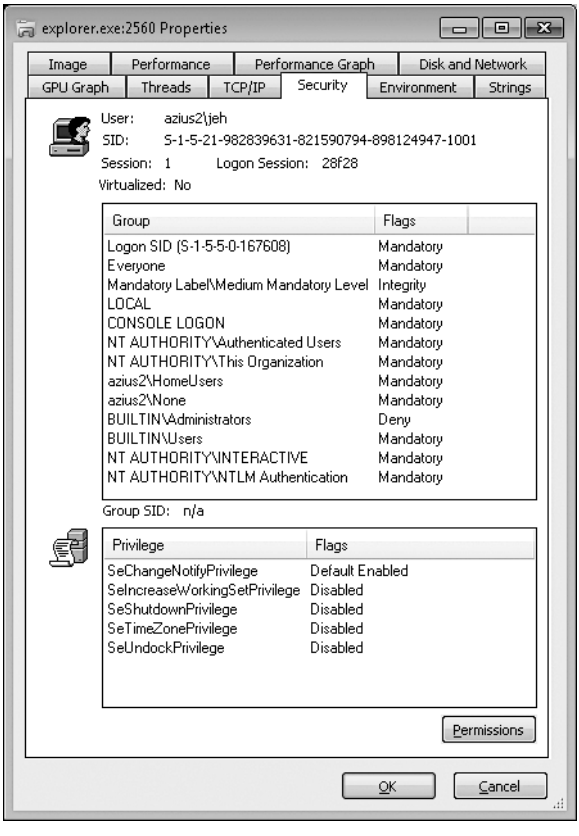
И наконец, изучив RID своей собственной учетной записи, вы узнаете, по крайней мере, количество учетных записей безопасности (которое равно результату вычитания числа 999 из значения вашего RID), созданных в ва-

шем домене или на вашей локальной машине (в зависимости от того, что вами используется, учетная запись домена или учетная запись локальной машины). Чтобы определить, каким учетным записям были назначены RID-идентификаторы, можно передать SID с RID, который нужно запросить утилите PsGetSid. Если PsGetSid сообщит, что сопоставить SID с каким-нибудь именем учетной записи не удалось и значение RID меньше, чем у вашей учетной записи, вы будете знать, что учетная запись, которой был назначен такой RID, была удалена.

Например, чтобы найти имя, назначенное учетной записи с RID, имеющим двадцать восьмой порядковый номер, передайте утилите PsGetSid SID домена, дополненный строковым значением –1027:

```
c:\>psgetsid S-1-5-21-1787744166-3910675280-2727264193-1027
Account for S-1-5-21-1787744166-3910675280-2727264193-1027:
User: redmond\daryl
```

Process Explorer в своей вкладке Security (Безопасность) также может показать вам информацию о SID-идентификаторах учетных записей и групп. В этой вкладке показывается информация о владельце данного процесса и к каким группам принадлежит учетная запись. Для просмотра этой информации нужно просто дважды щелкнуть кнопкой мыши на любом процессе (например, на Explorer.exe) в списке процессов, а затем щелкнуть на вкладке Security (Безопасность). Тогда вы сможете увидеть что-либо, подобное следующему изображению.



Информация, показанная в поле User (Пользователь), содержит легко читаемое имя учетной записи, владеющей этим процессом, а информация в поле SID содержит текущее значение SID-идентификатора. В список Group (Группа) включена информация о всех группах, в которые входит данная учетная запись. (Группы будут рассмотрены в данной главе чуть позже.) ■

Уровни целостности

Как уже ранее упоминалось, уровни целостности могут заменить разграничительный доступ, чтобы провести различия между процессом и объектами, запущенными от имени одного и того же пользователя и находящимися в его владении, предоставляя возможность изоляции кода и данных в рамках учетной записи пользователя. Механизм мандатного контроля целостности — mandatory integrity control (MIC) — позволяет SRM располагать более подробной информацией о природе вызывающего процесса путем его ассоциации с уровнем целостности. Он также предоставляет информацию о доверии, необходимую для доступа к объекту путем определения для него уровня целостности.

Эти уровни целостности определяются с помощью SID. Хотя уровни целостности могут иметь произвольные значения, в системе используются пять основных уровней для отделения друг от друга уровней привилегий. Описание этих уровней дано в табл. 6.3.

Таблица 6.3. SID-идентификаторы уровней целостности

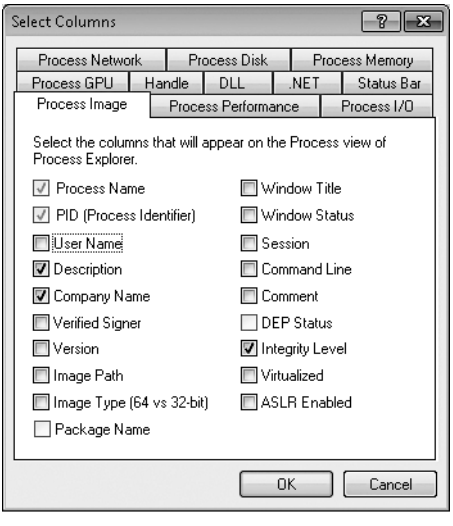
SID	Имя (Уровень)	Использование
S-1-16-0x0	Untrusted (0) (ненадежный)	Используется процессами, запущенными группой Anonymous. Он блокирует большинство доступов по записи
S-1-16-0x1000	Low (1) (низкий)	Используется защищенным режимом Internet Explorer. Он блокирует доступ по записи к большинству объектов системы (таких как файлы и разделы реестра)
S-1-16-0x2000	Medium (2) (средний)	Используется обычными приложениями, запущенными при включенной системе UAC
S-1-16-0x3000	High (3) (высокий)	Используется административными приложениями, запущенными через повышение уровня полномочий при включенной системе UAC, или обычными приложениями при выключенной системе UAC и при наличии у пользователя прав администратора
S-1-16-0x4000	System (4) (системный)	Используется службами и другими приложениями системного уровня (например, Wininit, Winlogon, Smss и т. д.)

ЭКСПЕРИМЕНТ: ПРОСМОТР УРОВНЕЙ ЦЕЛОСТНОСТИ ПРОЦЕССОВ

Для быстрого вывода уровней целостности для процессов на вашей системе можно воспользоваться средством Process Explorer из набора Sysinternals.

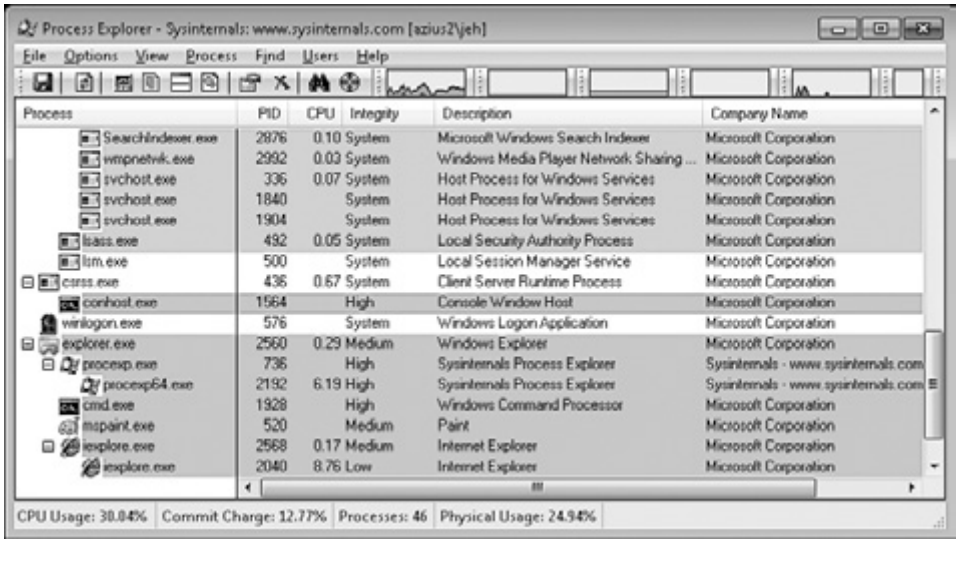
1. Запустите Internet Explorer в защищенном режиме.
2. Откройте в режиме повышенных привилегий окно командной строки.
3. Откройте в обычном режиме окно программы Microsoft Paint (не пользуясь повышенными привилегиями).

4. Теперь откройте Process Explorer, щелкните правой кнопкой мыши на любом из столбцов в списке процессов, а затем щелкните на пункте Select Columns (Выбрать столбцы). Вы должны увидеть диалоговое окно, подобное тому, что показано на следующем рисунке.



5. Установите флажок Integrity Level (Уровень целостности) и щелкните на кнопке OK, чтобы закрыть окно и сохранить изменения.
6. Теперь Process Explorer покажет вам уровень целостности процессов на вашей системе.

Для процесса защищенного режима Internet Explorer должен быть показан уровень Low (низкий), для Microsoft Paint уровень Medium (средний) и для окна командной строки, запущенного с повышенной привилегией, уровень High (высокий). Также следует заметить, что службы и системные процессы выполняются на еще более высоком уровне целостности System (системный).



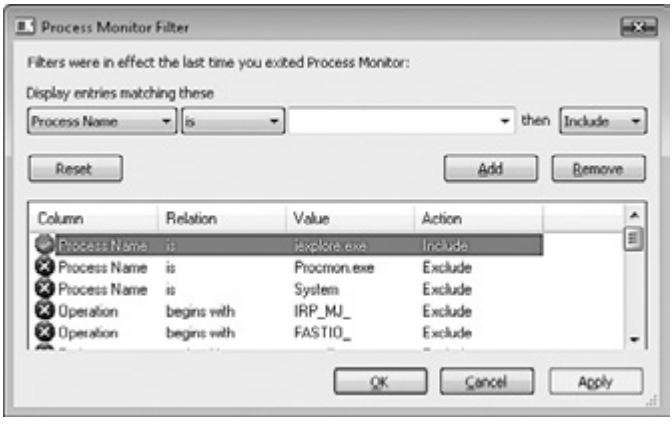
У каждого процесса есть уровень целостности, который представлен в маркере процесса и распространяется в соответствии со следующими правилами:

- Процесс обычно наследует уровень целостности своего родителя (это означает, что окно командной строки, запущенное с повышенными привилегиями, будет порождать другие процессы с повышенными привилегиями).
- Если файловый объект для исполняемого образа, которому принадлежит дочерний процесс, имеет уровень целостности и уровень целостности родительского процесса имеет значение medium (средний) или выше, дочерний процесс унаследует тот из двух уровней, который будет иметь более низкое значение.
- Родительский процесс может создать дочерний процесс с явно заданным уровнем целостности со значением, которое ниже значения его собственного уровня (например, при запуске защищенного режима Internet Explorer из окна командной строки, работающего с повышенными привилегиями). Для этого им используется функция DuplicateTokenEx, чтобы продублировать его собственный маркер доступа, а для изменения уровня целостности в новом маркере на желаемый им используется функция SetTokenInformation, а затем с этим новым маркером вызывается функция CreateProcessAsUser.

ЭКСПЕРИМЕНТ: ИЗУЧЕНИЕ ЗАЩИЩЕННОГО РЕЖИМА INTERNET EXPLORER

Как уже упоминалось, одним из пользователей механизма целостности Windows является защищенный режим Internet Explorer, который также называется Protected Mode Internet Explorer (PMIE). Это свойство было добавлено в Internet Explorer 7 для использования уровней целостности Windows. Этот эксперимент покажет вам, как PMIE использует уровни целостности для обеспечения безопасного пользования Интернетом. Для отслеживания поведения Internet Explorer воспользуемся утилитой Process Monitor.

1. Убедитесь в том, что на вашей системе не выключены UAC и PMIE (оба этих средства включены по умолчанию), и закройте все выполняемые экземпляры Internet Explorer.
2. Запустите Process Monitor и выберите в нем пункты меню Filter (Фильтр) для вывода диалогового окна настройки фильтра. Добавьте включающий фильтр для процесса Iexplore.exe, как показано на следующем рисунке.



3. Запустите утилиту Process Explorer и повторите действия из предыдущего эксперимента, чтобы вывести столбец Integrity Level (Уровень целостности).
4. Теперь запустите Internet Explorer. Вы должны увидеть шквал событий, появляющихся в окне Process Monitor, и быстрое развитие событий в Process Explorer, показывающее запуск и завершение некоторых процессов.

Как только Internet Explorer начнет выполняться, Process Explorer покажет вам два новых процесса Iexplore.exe, родительский Iexplore.exe, выполняемый на уровне целостности medium (средний), и его дочерние процессы, выполняемые на уровне целостности low (низкий).

Частью дополнительной защиты, предлагаемой PMIE, является то, что процессы Iexplore.exe, получающие доступ к веб-сайтам, выполняются на уровне целостности low (низкий). Поскольку Internet Explorer размещает вкладки в нескольких процессах, при создании дополнительных вкладок вы можете увидеть дополнительные экземпляры Iexplore.exe. Есть один родительский процесс Iexplore.exe, работающий в качестве посредника, который предоставляет доступ к частям системы и недоступный тем процессам, которые выполняются на уровне целостности low (низкий) для того, чтобы, к примеру, сохранить или открыть файлы из других частей файловой системы. ■

В табл. 6.3 перечислены уровни целостности, связанные с процессами, а как насчет объектов? Объекты также имеют уровень целостности, сохраненный как часть их дескриптора безопасности, в структуре под названием мандатный ярлык — mandatory label.

Для поддержки миграции из предыдущих версий Windows (тех, чьи разделы реестра и файлы не будут включать информации об уровне целостности), а также для упрощения работы разработчиков приложений все объекты имеют подразумеваемый уровень целостности, дабы избежать его конкретного указания. Этот подразумеваемый уровень целостности имеет значение medium (средний), означающий, что мандатная политика (которая вскоре будет рассмотрена) в отношении объекта будет выполняться на маркерах, связывающих этот объект с уровнем целостности, меньшим, чем medium (средний).

Когда процесс создает объект без указания уровня целостности, система проверяет уровень целостности в маркере. Для маркеров с уровнем целостности medium или выше подразумеваемое значение уровня целостности объекта остается medium. Но когда маркер содержит уровень целостности ниже medium, объект создается с подразумеваемым уровнем целостности, соответствующем уровню в маркере.

Смысл того, что объекты, создаваемые процессами, имеющими уровни целостности high или system, сами по себе имеют уровень целостности medium, заключается в том, что пользователи могут выключать и включать систему UAC. Если уровень целостности объекта всегда бы наследовал уровень целостности своего создателя, приложения администратора, выключившего UAC, а затем снова включившего эту систему, потенциально получали бы отказ, поскольку администратор не смог бы изменить какие-либо настройки реестра или файлов, созданных при выполнении на высоком уровне целостности (high). У объектов также может быть явно заданный уровень целостности, устанавливаемый систе-

мой или создателем объекта. Например, следующим объектам при их создании ядром задается явный уровень целостности:

- процессы;
- потоки;
- маркеры;
- задания.

Причиной присваивания уровня целостности этим объектам является предотвращение со стороны процесса того же пользователя, но запущенного на более низком уровне целостности доступа к таким объектам и изменения их контекста или поведения (например, DLL-инъекции или изменения кода).

ЭКСПЕРИМЕНТ: ПРОСМОТР УРОВНЯ ЦЕЛОСТНОСТИ ОБЪЕКТОВ

Для вывода уровня целостности имеющихся в системе таких объектов, как файлы, процессы и разделы реестра, можно воспользоваться утилитой Accesschk из набора Sysinternals. Проведем эксперимент, в котором показывается назначение в Windows каталога LocalLow.

1. Перейдите в окне командной строки в каталог C:\Users\UserName\.
2. Попробуйте запустить утилиту Accesschk в отношении папки AppData:

```
C:\Users\UserName> accesschk -v appdata
```

3. Обратите внимание на разницу между Local и LocalLow в выводимой информации, которая должна быть похожа на следующую:

```
C:\Users\UserName\AppData\Local
  Medium Mandatory Level (Default) [No-Write-Up]
  [...]C:\Users\UserName\AppData\LocalLow
  Low Mandatory Level [No-Write-Up]
  [...]
C:\Users\UserName\AppData\Roaming
  Medium Mandatory Level (Default) [No-Write-Up]
  [...]
```

4. Заметьте, что у каталога LocalLow имеется уровень целостности, установленный в Low, а у каталогов Local и Roaming уровень целостности имеет значение Medium (Default). Слово «default» означает, что система использует подразумеваемый уровень целостности.
5. Чтобы утилита Accesschk выводила только подразумеваемые уровни целостности, ей можно передать флаг –e. Если ее снова запустить в отношении папки AppData, вы заметите, что будет выведена только информация о папке LocalLow.

Ключи –o (Object, объект), –k (Registry Key, раздел реестра) и –p (Process, процесс) позволяют указывать объекты, отличные от файла или каталога. ■

Кроме уровней целостности у объектов есть также мандатная политика, определяющая действующий уровень защиты, применяемый на основе проверки уровня целостности. Три возможных типа такой политики показаны в табл. 6.4. Уровень целостности и мандатная политика хранятся вместе в одном и том же ACE.

Таблица 6.4. Мандатные политики объекта

Политика	Объекты, в которых она присутствует по умолчанию	Описание
No-Write-Up (отказ в записи)	Подразумевается на всех объектах	Используется для ограничения доступа к объекту по записи со стороны процессов, имеющих более низкий уровень целостности
No-Read-Up (отказ в чтении)	Только на объектах процессов	Используется для ограничения доступа к объекту по чтению со стороны процессов, имеющих более низкий уровень целостности. Конкретное использование в отношении объектов процессов создает защиту от утечки информации путем блокирования чтения адресного пространства из внешнего процесса
No-Execute-Up (отказ в выполнении)	Только на двоичных реализациях COM-классов	Используется для ограничения доступа к объекту по выполнению со стороны процессов, имеющих более низкий уровень целостности. Конкретное использование в отношении COM-классов преследует цель ограничения прав на запуск и активацию COM-класса

Маркеры

Для идентификации контекста безопасности процесса или потока SRM использует объект под названием маркер (или маркер доступа). Контекст безопасности состоит из информации, описывающей учетную запись, группы и привилегии, связанные с процессом или потоком. Маркеры также включают такую информацию, как ID сеанса, уровень целостности и состояние виртуализации UAC. (Привилегии и механизм виртуализации UAC будут рассмотрены в данной главе чуть позже.)

При выполнении процесса входа в систему (рассматриваемого в конце данной главы) процесс LSASS создает исходный маркер для представления пользователя, входящего в систему. Затем он определяет, относится ли пользователь к группе, имеющей высокие полномочия, или обладает ли он высокими привилегиями. На данном этапе проверка принадлежности к группам ведется в следующем порядке:

- Built-In Administrators (Встроенная группа администраторов);
- Certificate Administrators (Администраторы сертификатов);
- Domain Administrators (Администраторы доменов);
- Enterprise Administrators (Администраторы предприятий);
- Policy Administrators (Администраторы политик);
- Schema Administrators (Администраторы схем);
- Domain Controllers (Контроллеры доменов);
- Enterprise Read-Only Domain Controllers (Контролеры принадлежащих предприятию доменов, предназначенных только для чтения);
- Read-Only Domain Controllers (Контролеры доменов, предназначенных только для чтения);
- Account Operators (Операторы учетных записей);

- Backup Operators (Операторы по созданию резервных копий);
- Cryptographic Operators (Операторы криптографических систем);
- Network Configuration Operators (Операторы сетевой конфигурации);
- Print Operators (Операторы вывода на печать);
- System Operators (Системные операторы);
- RAS Servers (Серверы служб удаленного доступа — RAS);
- Power Users (Пользователи с повышенными привилегиями);
- Pre-Windows 2000 Compatible Access (Доступ в режиме совместимости с системами, предшествовавшими Windows 2000).

Многие из перечисленных групп используются только на системах с объединенными доменами и не дают пользователям локальных административных прав напрямую. Вместо этого они позволяют пользователям изменять настройки, распространяющиеся на весь домен.

Проверяется также наличие следующих привилегий:

- SeBackupPrivilege (на создание резервной копии);
- SeCreateTokenPrivilege (на создание маркера);
- SeDebugPrivilege (на отладку);
- SeImpersonatePrivilege (на работу от имени других пользователей);
- SeLabelPrivilege (на создание меток);
- SeLoadDriverPrivilege (на загрузку драйверов);
- SeRestorePrivilege (на восстановление системы);
- SeTakeOwnershipPrivilege (на смену владельца);
- SeTcbPrivilege (на работу с блоком управления потоком).

Более подробно эти привилегии рассмотрены в одном из следующих разделов.

Если установлена принадлежность к одной и более из этих групп или наличие одной и более привилегий, LSASS создает для пользователя маркер с ограничениями (также называемый фильтрованным административным маркером) и создает сеанс входа в систему. Обычный маркер пользователя прикрепляется к исходному процессу или процессам, запускаемым Winlogon (по умолчанию Userinit.exe).

ПРИМЕЧАНИЕ

Если система UAC выключена, администраторы работают с маркером, включающим их членство в административных группах и привилегии.

Поскольку по умолчанию дочерние процессы наследуют копию маркера своего создателя, все процессы в сеансе пользователя запускаются под одним и тем же маркером. Маркер можно также сгенерировать, используя Windows-функцию LogonUser. Затем этот маркер можно использовать для создания процесса, выполняющегося в контексте безопасности пользователя, вошедшего в систему посредством функции LogonUser, передав маркер Windows-функции CreateProcessAsUser. Функция CreateProcessWithLogon сочетает все это в одном вызове, именно так команда Runas запускает процессы под альтернативными маркерами.

Маркеры варьируются по длине, поскольку у разных учетных записей пользователей имеются разные наборы привилегий и связанные с ними учетные

записи групп. Но все маркеры содержат одинаковые типы информации. Наиболее важное содержимое маркера представлено на рис. 6.4.

Для определения доступности объектов и вида защищаемых операций имеющиеся в Windows механизмы безопасности используют два компонента. Один компонент включает в себя принадлежащий маркеру SID учетной записи пользователя и поля SID групп. Монитор безопасности — security reference monitor (SRM) использует SID-идентификаторы для определения, может ли процесс или поток получить запрашиваемый доступ к защищаемому объекту, например к NTFS-файлу.

Имеющиеся в маркере групповые SID-идентификаторы показывают, в какие группы входит учетная запись пользователя. Например, серверное приложение может запретить конкретные группы для ограничения полномочий маркера, когда серверное приложение выполняет действия, запрещенные клиентам. Запрет группы имеет примерно такой же эффект, как если бы группа не было представлена в маркере[1]. Групповые SID-идентификаторы могут также включать специальный SID, содержащий уровень целостности процесса или потока. SRM использует еще одно поле в маркере, дающее описание мандатной политики целостности, чтобы выполнять рассматриваемую далее мандатную проверку целостности.

Второй компонент маркера, который определяет, что может поток или процесс с этим маркером делать, называется массивом привилегий. Он представляет собой список прав, связанных с маркером. В качестве примера привилегии можно назвать право процесса или потока, связанного с маркером, выключать компьютер. Более подробное рассмотрение привилегий дается в этой главе чуть позже.

Рис. 6.4. Маркер доступа

Имеющееся в маркере исходное поле основной группы и исходный список управления избирательным доступом — discretionary access control list (DACL) — являются теми атрибутами безопасности, которые Windows применяет к объектам, создаваемым процессом или потоком при использовании данного маркера. За счет включения в маркеры информации безопасности Windows упрощает для процесса или потока создание объектов со стандартными атрибутами безопасности, поскольку процесс или поток не нуждается в запросе отдельной информации безопасности для каждого создаваемого им объекта.

[1] Это приводит к появлению группы только для запрета, рассматриваемой чуть позже. Выключенные SID-идентификаторы используются в качестве части проверок безопасности доступа, рассматриваемых далее в этой главе.

Каждый тип маркера устанавливает различие между основным маркером, идентифицирующим контекст безопасности процесса, и маркером заимствования (который является таким типом маркера, который потоки используют для временного заимствования другого контекста безопасности, обычно для другого пользователя). Маркеры заимствования содержат уровень заимствования, который обозначает тот тип заимствования, который активен в маркере (заимствование прав будет рассмотрено в этой главе чуть позже).

Маркер также включает мандатную политику для процесса или потока, которая определяет, как мандатный контроль целостности (MIC) будет вести себя при обработке этого маркера. Есть две политики:

- TOKEN_MANDATORY_NO_WRITE_UP, которая включается по умолчанию, устанавливает для маркера политику No-Write-Up (отказ в записи), указывающую, что данный процесс или поток не сможет иметь допуск по записи к объектам с более высоким уровнем целостности.
- TOKEN_MANDATORY_NEW_PROCESS_MIN, которая также включается по умолчанию, указывает, что SRM будет искать уровень целостности исполняемого образа при запуске дочернего процесса и вычислять минимальный уровень целостности родительского процесса и уровень целостности файлового объекта в качестве уровня целостности дочернего процесса.

Флаги маркеров включают параметры, которые определяют поведение конкретных механизмов UAC и UIPI, например доступа к виртуализации и к пользовательскому интерфейсу. Эти механизмы будут рассмотрены в этой главе чуть позже.

Каждый маркер может также содержать атрибуты, назначаемые службой идентификации приложений — Application Identification (частью AppLocker), когда правила AppLocker уже определены. AppLocker и использование им атрибутов в маркере доступа рассматриваются в этой главе чуть позже.

Все остальные поля в маркере служат информационным целям. Поле источника маркера (source) содержит краткое текстовое описание создателя маркера. Программы, желающие знать о происхождении маркера, используют источник маркера, чтобы отличить друг от друга такие источники, как диспетчер сеансов —Windows Session Manager, сетевой файловый сервер или сервер удаленного вызова процедуры — remote procedure call (RPC). Идентификатор маркера является локально уникальным идентификатором — locally unique identifier (LUID), — который SRM присваивает маркеру при его создании. Исполняющая система Windows обслуживает свой LUID (executive LUID), монотонно возрастающий счетчик, который используется ею для назначения уникального цифрового идентификатора каждому маркеру. LUID гарантирует уникальность только до тех пор, пока система не будет выключена.

Еще одной разновидностью LUID является ID аутентификации. Создатель маркера назначает ему ID аутентификации при вызове функции LsaLogonUser. Если создатель не указал LUID, LSASS получает LUID из LUID исполняющей системы. LSASS копирует ID аутентификации для всех маркеров, происходящих от исходного маркера входа в систему. Программа может получить ID аутентификации маркера, чтобы посмотреть, не принадлежит ли маркер к тому же сеансу входа в систему, что и другие маркеры экзаменующей программы.

При каждом изменении характеристик маркера ID модификации обновляется за счет LUID исполняющей системы. Приложение может проверять ID модификации для обнаружения изменений в контексте безопасности со времени последнего использования контекста.

ПРИМЕЧАНИЕ

Чтобы гарантировать безопасность системы, поля в маркере являются неизменными (поскольку они находятся в памяти ядра). За исключением полей, которые могут быть изменены через определенные системные вызовы, предназначенные для модификации конкретных атрибутов маркера (при условии наличия у вызывающего процесса соответствующих прав доступа к объекту маркера), такие данные, как привилегии и SID-идентификаторы, в маркере не могут быть изменены из пользовательского режима ни при каких условиях.

Маркеры содержат поле истечения срока действия, которое может использоваться приложениями, выполняющими свои собственные мероприятия безопасности по отклонению маркера после определенного количества времени. Но сама система Windows не заставляет указывать время истечения срока действия маркера.

ЭКСПЕРИМЕНТ: ПРОСМОТР МАРКЕРОВ ДОСТУПА

Команда отладчика ядра dt _TOKEN выводит формат внутреннего объекта маркера. Хотя эта структура отличается от структуры маркера пользовательского режима, возвращается функциями безопасности Windows API, их поля совпадают. Дополнительные сведения о маркерах можно найти в документации по Windows SDK.

Вывод, полученный в результате выполнения команды отладчика ядра dt nt!_TOKEN, имеет следующий вид:

```
kd> dt nt!_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
   +0x040 Privileges           : _SEP_TOKEN_PRIVILEGES
   +0x058 AuditPolicy          : _SEP_AUDIT_POLICY
   +0x074 SessionId            : Uint4B
   +0x078 UserAndGroupCount    : Uint4B
   +0x07c RestrictedSidCount   : Uint4B
   +0x080 VariableLength       : Uint4B
   +0x084 DynamicCharged       : Uint4B
   +0x088 DynamicAvailable     : Uint4B
   +0x08c DefaultOwnerIndex    : Uint4B
   +0x090 UserAndGroups        : Ptr32 _SID_AND_ATTRIBUTES
   +0x094 RestrictedSids       : Ptr32 _SID_AND_ATTRIBUTES
   +0x098 PrimaryGroup         : Ptr32 Void
```

продолжение ⇩

```
   +0x09c DynamicPart              : Ptr32 Uint4B
   +0x0a0 DefaultDacl              : Ptr32 _ACL
   +0x0a4 TokenType                : _TOKEN_TYPE
   +0x0a8 ImpersonationLevel       : _SECURITY_IMPERSONATION_LEVEL
   +0x0ac TokenFlags               : Uint4B
   +0x0b0 TokenInUse               : UChar
   +0x0b4 IntegrityLevelIndex      : Uint4B
   +0x0b8 MandatoryPolicy          : Uint4B
   +0x0bc ProxyData                : Ptr32 _SECURITY_TOKEN_PROXY_DATA
   +0x0c0 AuditData                : Ptr32 _SECURITY_TOKEN_AUDIT_DATA
   +0x0c4 LogonSession             : Ptr32 _SEP_LOGON_SESSION_REFERENCES
   +0x0c8 OriginatingLogonSession  : _LUID
   +0x0d0 SidHash                  : _SID_AND_ATTRIBUTES_HASH
   +0x158 RestrictedSidHash        : _SID_AND_ATTRIBUTES_HASH
   +0x1e0 VariablePart             : Uint4B
```

Маркер процесса можно изучить с помощью команды !token. Адрес маркера можно найти в выводе команды !process:

```
lkd> !process d6c 1
Searching for Process with Cid == d6c
PROCESS 85450508  SessionId: 1  Cid: 0d6c    Peb: 7ffda000  ParentCid: 0ecc
    DirBase: cc9525e0  ObjectTable: afd75518  HandleCount:  18.
    Image: cmd.exe
    VadRoot 85328e78 Vads 24 Clone 0 Private 148. Modified 0. Locked 0.
    DeviceMap a0688138
    Token                             afd48470
    ElapsedTime                       01:10:14.379
    UserTime                          00:00:00.000
    KernelTime                        00:00:00.000
    QuotaPoolUsage[PagedPool]         42864
    QuotaPoolUsage[NonPagedPool]      1152
    Working Set Sizes (now,min,max)   (566, 50, 345) (2264KB, 200KB, 1380KB)
    PeakWorkingSetSize                582
    VirtualSize                       22 Mb
    PeakVirtualSize                   25 Mb
    PageFaultCount                    680
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      437

lkd> !token afd48470
_TOKEN afd48470
TS Session ID: 0x1
User: S-1-5-21-2778343003-3541292008-524615573-500 (User: ALEX-LAPTOP\Administrator)
Groups:
 00 S-1-5-21-2778343003-3541292008-524615573-513 (Group: ALEX-LAPTOP\None)
    Attributes - Mandatory Default Enabled
 01 S-1-1-0 (Well Known Group: localhost\Everyone)
```

```
    Attributes - Mandatory Default Enabled
02 S-1-5-21-2778343003-3541292008-524615573-1000 (Alias: ALEX-LAPTOP\Debugger Users)
    Attributes - Mandatory Default Enabled
 03 S-1-5-32-544 (Alias: BUILTIN\Administrators)
    Attributes - Mandatory Default Enabled Owner
 04 S-1-5-32-545 (Alias: BUILTIN\Users)
    Attributes - Mandatory Default Enabled
 05 S-1-5-4 (Well Known Group: NT AUTHORITY\INTERACTIVE)
    Attributes - Mandatory Default Enabled
 06 S-1-5-11 (Well Known Group: NT AUTHORITY\Authenticated Users)
    Attributes - Mandatory Default Enabled
 07 S-1-5-15 (Well Known Group: NT AUTHORITY\This Organization)
    Attributes - Mandatory Default Enabled
 08 S-1-5-5-0-89263 (no name mapped)
    Attributes - Mandatory Default Enabled LogonId
 09 S-1-2-0 (Well Known Group: localhost\LOCAL)
    Attributes - Mandatory Default Enabled
 10 S-1-5-64-10 (Well Known Group: NT AUTHORITY\NTLM Authentication)
    Attributes - Mandatory Default Enabled
 11 S-1-16-12288 Unrecognized SID
    Attributes - GroupIntegrity GroupIntegrityEnabled
Primary Group: S-1-5-21-2778343003-3541292008-524615573-513 (Group: ALEX-LAPTOP\None)
Privs:
 05 0x000000005 SeIncreaseQuotaPrivilege          Attributes -
 08 0x000000008 SeSecurityPrivilege               Attributes -
 09 0x000000009 SeTakeOwnershipPrivilege          Attributes -
 10 0x00000000a SeLoadDriverPrivilege             Attributes -
 11 0x00000000b SeSystemProfilePrivilege          Attributes -
 12 0x00000000c SeSystemtimePrivilege             Attributes -
 13 0x00000000d SeProfileSingleProcessPrivilege   Attributes -
 14 0x00000000e SeIncreaseBasePriorityPrivilege   Attributes -
 15 0x00000000f SeCreatePagefilePrivilege         Attributes -
 17 0x000000011 SeBackupPrivilege                 Attributes -
 18 0x000000012 SeRestorePrivilege                Attributes -
 19 0x000000013 SeShutdownPrivilege               Attributes -
 20 0x000000014 SeDebugPrivilege                  Attributes -
 22 0x000000016 SeSystemEnvironmentPrivilege      Attributes -
 23 0x000000017 SeChangeNotifyPrivilege           Attributes - Enabled Default
 24 0x000000018 SeRemoteShutdownPrivilege         Attributes -
 25 0x000000019 SeUndockPrivilege                 Attributes -
 28 0x00000001c SeManageVolumePrivilege           Attributes -
 29 0x00000001d SeImpersonatePrivilege            Attributes - Enabled Default
 30 0x00000001e SeCreateGlobalPrivilege           Attributes - Enabled Default
 33 0x000000021 SeIncreaseWorkingSetPrivilege     Attributes -
 34 0x000000022 SeTimeZonePrivilege               Attributes -
 35 0x000000023 SeCreateSymbolicLinkPrivilege     Attributes -
```

продолжение ⇩

```
Authentication ID:          (0,be1a2)
Impersonation Level:        Identification
TokenType:                  Primary
Source: User32              TokenFlags: 0x0 ( Token in use )
Token ID: 711076            ParentToken ID: 0
Modified ID:                (0, 711081)
RestrictedSidCount: 0       RestrictedSids: 00000000
OriginatingLogonSession: 3e7
```

Опосредованно просмотреть содержимое маркера можно с помощью вкладки Security (Безопасность) утилиты Process Explorer в относящемся к процессу диалоговом окне Properties (Свойства). В этом окне будут показаны группы и привилегии, включенные в маркер изучаемого процесса. ■

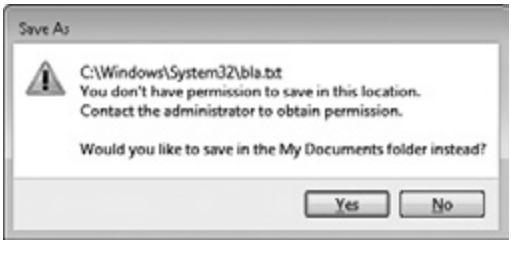
ЭКСПЕРИМЕНТ: ЗАПУСК ПРОГРАММЫ НА УРОВНЕ ЦЕЛОСТНОСТИ LOW (НИЗКИЙ)

Когда программа запускается с повышенными привилегиями, либо путем использования пункта Запуск от имени администратора (Run As Administrator), либо потому что программа этого требует, программа явным образом запускается на высоком уровне целостности, но есть также возможность запустить программу (но не PMIE) на низком уровне целостности, используя утилиту Psexec из набора Sysinternals:

1. Запустите программу Блокнот (Notepad) на низком уровне целостности, используя следующую команду:

```
c:\psexec –l notepad.exe
```

2. Попробуйте открыть файл (например, один из файлов с расширением .XML) в каталоге %SystemRoot%\System32. Обратите внимание на то, что вы можете просматривать каталог и открывать любой, содержащийся в нем файл.
3. Теперь воспользуйтесь командой меню программы Блокнот Файл (File)▸ Создать (New), введите какой-нибудь текст в окно и попробуйте сохранить его в каталоге %SystemRoot%\System32. Блокнот должен вывести окно сообщения об отсутствии прав и рекомендацией сохранить файл в папке Документы (Documents).
4. Воспользуйтесь предложением программы Блокнот. Вы снова получите аналогичное окно, и так будет при каждой подобной попытке.



5. Теперь попробуйте сохранить файл в каталоге LocalLow вашего профиля пользователя, показанный в эксперименте, проделанном ранее в данной главе.

В предыдущем эксперименте сохранение файла в каталоге LocalLow сработает, поскольку программа Блокнот была запущена с низким уровнем целостности, и только у каталога LocalLow также имеется низкий уровень целостности. Все остальные места, в которые вы пробовали записать файл, имели подразумеваемый уровень целостности medium (средний). (Вы можете проверить этот факт с помощью утилиты Accesschk.) Но чтение из каталога %SystemRoot%\System32, равно как и открытие находящихся в нем файлов, работало даже при том, что каталог и его файл также имели подразумеваемый средний уровень целостности. ■

Заимствование прав

Заимствование прав (impersonation) является весьма эффективным свойством Windows, часто используемым в ее модели безопасности. Windows также использует заимствование прав в своей модели программирования клиент-сервер. Например, серверное приложение может предоставить доступ к таким ресурсам, как файлы, принтеры или базы данных. Клиенты ждут доступа к ресурсу, отправив запрос на сервер. Когда сервер получает запрос, он должен убедиться, что у клиента есть разрешение на осуществление желаемой операции над ресурсом. Например, если пользователь на удаленной машине пробует удалить файл на общей NTFS-системе, сервер, экспортирующий общий ресурс, должен определить, разрешено ли пользователю удалять файл. Явным образом определить наличие разрешения у пользователя сервер может путем запроса SID-идентификаторов учетной записи и групп пользователя и сканирования атрибутов безопасности файла. Такой подход для программы слишком сложен, при его реализации нетрудно наделать ошибок, и он не допускает прозрачной поддержки новых свойств системы безопасности. Поэтому для облегчения работы сервера Windows предоставляет службы заимствования прав.

Заимствования прав позволяет серверу уведомить SRM о том, что сервер временно заимствует профиль безопасности клиента, запрашивающего ресурс. Затем сервер может получить доступ к ресурсам от имени клиента, а проверку безопасности берет на себя SRM, но делает это на основе контекста безопасности того клиента, права которого были позаимствованы. Обычно у сервера имеется доступ к более широкому спектру ресурсов чем у клиента, и в ходе заимствования прав он теряет некоторые из своих полномочий безопасности. Но может быть верным и обратное утверждение: в ходе заимствования прав сервер может получить полномочия безопасности.

Сервер заимствует права клиента только в рамках того потока, который делает запрос на заимствование прав. Структура данных управления потоком содержит дополнительную запись для маркера заимствования прав. Но основной маркер потока, представляющий его реальные полномочия безопасности, всегда доступен в структуре управления, принадлежащей процессу.

Windows делает доступным заимствование прав посредством нескольких механизмов. Например, если сервер обменивается данными с клиентом через именованный канал, сервер может воспользоваться функцией ImpersonateNamedPipeClient, входящей в Windows API, чтобы сообщить SRM о том, что он хочет позаимствовать права пользователя на другом конце канала. Если сервер связан с клиентом через динамический обмен данными —

Dynamic Data Exchange (DDE) — или через RPC, он межет сделать такие же запросы на заимствование прав с помощью функции DdeImpersonateClient и функции RpcImpersonateClient. С помощью функции ImpersonateSelf поток может создать маркер заимствования прав, являющийся простой копией маркера его процесса. Затем поток может изменить его маркер заимствования прав, возможно, для отключения SID-идентификаторов или привилегий. Пакет интерфейса поддержки безопасности поставщика — Security Support Provider Interface (SSPI) — может заимствовать права своих клиентов с помощью функции ImpersonateSecurityContext. SSPI-пакеты реализуют сетевой протокол аутентификации, например LAN Manager версии 2 или Kerberos. Другие интерфейсы, такие как COM, экспонируют заимствование прав через собственные API-функции, например через функцию CoImpersonateClient.

После завершения потоком сервера своей задачи он возвращается к своему основному контексту безопасности. Эти форма заимствования прав удобны для выполнения конкретных действий по запросу клиента и для обеспечения правильной проверки доступа к объекту. (Например, осуществляемая проверка дает идентичность клиента, чьи права заимствуются, а не идентичность серверного процесса.) Недостатком таких форм заимствования прав является невозможность выполнения всей программы в контексте клиента. Кроме того, маркер заимствования не может дать доступ к файлам или принтерам на общих сетевых ресурсах, если права не позаимствованы на уровне делегирования (которое будет вскоре рассмотрено) и не имеются соответствующие полномочия на аутентификацию на удаленной машине или если общий файл или принтер не поддерживает нулевые сеансы. (Нулевой сеанс является результатом анонимного входа в систему.)

Если в контексте безопасности клиента должно выполняться все приложение или должен быть получен доступ к сетевым ресурсам без использования заимствования прав, клиент должен быть зарегистрирован в системе. Это действие позволяет выполнить функция LogonUser из состава Windows. Функции LogonUser в качестве входных данных передается имя учетной записи, пароль, домен или имя компьютера, тип входа в систему (например, интерактивный, пакетный или служебный) и поставщик входа в систему, а она возвращает основной маркер доступа. Поток сервера может позаимствовать маркер в качестве маркера заимствования, или же сервер может запустить программу, имеющую полномочия клиента, в качестве основного маркера доступа. С точки зрения безопасности, создаваемый процесс использует маркер, возвращенный из интерактивного входа в систему через функцию LogonUser, например, с API-функцией CreateProcessAsUser, подобно программе, запущенной пользователем, который зарегистрировался на машине в интерактивном режиме. Недостатком такого подхода является то, что сервер должен получить имя пользователя и пароль. Если серверу эта информация передается по сети, то она должна быть надежно зашифрована, чтобы злоумышленники, отслеживающие сетевой трафик, не могли ее перехватить.

Чтобы не допустить неправильного использования заимствования прав, Windows не позволяет серверам выполнять заимствование прав без согласия клиентов. Клиентский процесс может ограничить уровень заимствования прав, доступный для выполнения на серверном процессе, указав при подключении к серверу качество безопасности службы — security quality of service (SQOS). Например, при открытии именованного канала процесс может указать

для Windows-функции CreateFile флаги SECURITY_ANONYMOUS, SECURITY_IDENTIFICATION, SECURITY_IMPERSONATION или SECURITY_DELEGATION. Каждый уровень позволяет серверу выполнять различные типы операций с оглядкой на контекст безопасности клиента:

- SecurityAnonymous является самым ограниченным уровнем заимствования прав — сервер не может заимствовать права или идентифицировать клиента.
- SecurityIdentification позволяет серверу получать информацию об идентичности (SID-идентификаторы) клиента и привилегии клиента, но сервер не может заимствовать его права.
- SecurityImpersonation позволяет серверу идентифицировать клиента и заимствовать его права на локальной системе.
- SecurityDelegation — наиболее разрешающий уровень заимствования прав. Он позволяет сервреу заимствовать права клиента на локальной и удаленных системах.

Другие интерфейсы, такие как RPC, используют другие константы со сходными значениями (например, RPC_C_IMP_LEVEL_IMPERSONATE).

Если клиент не установил уровень заимствования прав, по умолчанию Windows выбирает уровень SecurityImpersonation. Функция CreateFile также принимает в качестве модификаторов настройки заимствования прав константы SECURITY_EFFECTIVE_ONLY и SECURITY_CONTEXT_TRACKING:

- SECURITY_EFFECTIVE_ONLY не позволяет серверу включать и отключать привилегии клиента или групп в ходе заимствования сервером их прав.
- SECURITY_CONTEXT_TRACKING указывает на то, что любые изменения, сделанные клиентом в отношении его контекста безопасности, воздействуют на сервер, позаимствовавший его права. Если эта настройка не указана, сервер заимствует контекст клиента на время заимствования прав и не принимает никаких изменений. Эта настройка принимается во внимание только когда процессы клиента и сервера находятся на одной и той же системе.

Для предотвращения сценариев имитации, в ходе которых процесс с низким уровнем целостности может создать пользовательский интерфейс, получающий полномочия пользователя, а затем использующий функцию LogonUser для получения маркера этого пользователя, применяется специальная политика целостности для сценариев заимствования прав. Поток не может заимствовать права на маркер с более высоким уровнем целостности, чем имеется у него самого. Например, приложение, имеющее низкий уровень целостности, не может имитировать диалоговое окно, запрашивающее административные полномочия с последующей попыткой запуска процесса с более высоким уровнем привилегий. Политика, использующая механизм целостности для маркеров доступа с заимствованием прав, заключается в том, что уровень целостности маркера доступа, возвращаемого функцией LsaLogonUser, не должен быть выше уровня целостности вызывавшего эту функцию процесса.

Ограниченные маркеры

Ограниченный маркер создается из основного маркера или маркера заимствования с использованием функции CreateRestrictedToken. Ограниченный маркер

является копией маркера, из которого он происходит, со следующими возможными изменениями:

- Из массива привилегий может быть удален ряд привилегий.
- SID-идентификаторы в маркере могут быть помечены как только для запрета (deny-only). Такие SID-идентификаторы удаляют доступ к любым ресурсам, для которых доступ по SID-идентификатору запрещен с помощью соответствующего запрещающего доступ ACE-элемента, который в противном случае был бы отменен ACE-элементом, предоставляющим доступ к группе, содержащей SID в дескрипторе безопасности чуть раньше.
- SID-идентификаторы в маркере могут быть помечены как ограниченные. Такие SID-идентификаторы подлежат повторному проходу в алгоритме проверки доступа, который будет подвергать разбору только ограниченные SID-идентификаторы маркера. В результате как первого, так и второго прохода доступ к ресурсу или объекту либо должен быть, либо не должен быть предоставлен.

Ограниченные маркеры применяются в том случае, когда приложению нужно позаимствовать права клиента на усеченном уровне безопасности, главным образом из соображений предосторожности при запуске кода, не внушающего доверия. Например, ограниченный маркер может иметь удаленную из него привилегию по завершении работы системы, чтобы не дать коду, выполняемому в контексте безопасности ограниченного маркера, перезапустить систему.

Фильтрованный маркер администратора

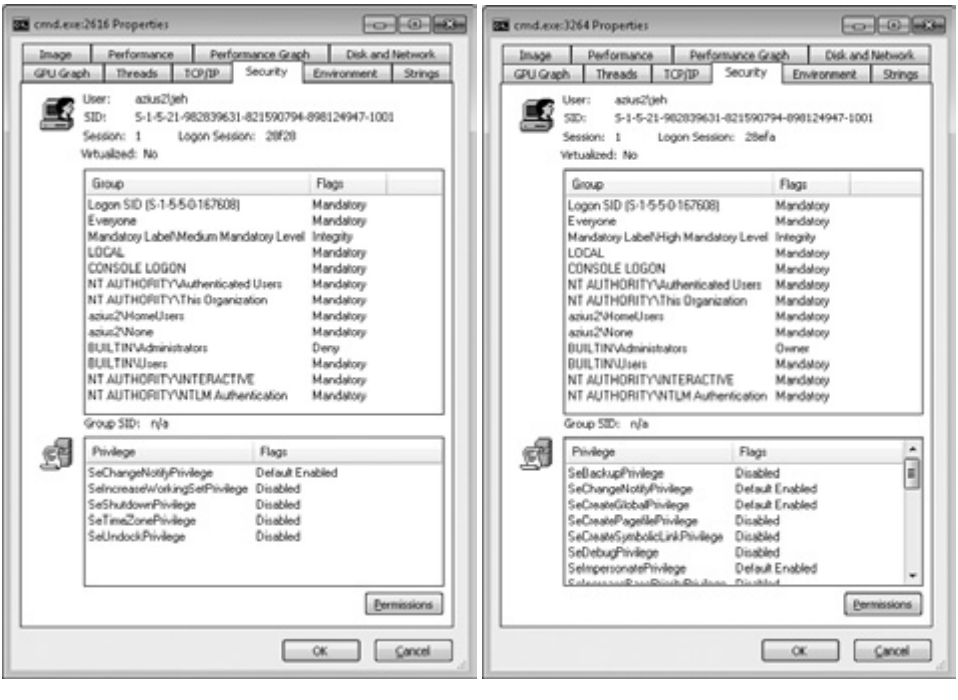
Как вы уже поняли, ограниченные маркеры также используются UAC для создания фильтрованного маркера администратора, который будет наследоваться всеми пользовательскими приложениями. Фильтрованный маркер администратора имеет следующие характеристики:

- Для уровня целостности устанавливается значение medium (средний).
- Упомянутые ранее администраторские и подобные администраторским SID-идентификаторы помечаются только для запрета, чтобы предотвращать прореху в системе безопасности, возникающую при удалении целиком всей группы. Например, пусть у файла был доступ к списку управления доступом — access control list (ACL), — который полностью отказывал группе «Администраторы» в доступе, но предоставлял некоторый доступ другой группе, к которой принадлежал пользователь. Тогда этот пользователь получил бы доступ, если группа «Администраторы» отсутствовала бы в маркере, что предоставило бы версии пользовательской идентификации обычного пользователя более широкий доступ, чем при административной идентификации пользователя.
- Удаляются все привилегии за исключением Change Notify, Shutdown, Undock, Increase Working Set и Time Zone.

ЭКСПЕРИМЕНТ: ПРОСМОТР ФИЛЬТРОВАННЫХ МАРКЕРОВ АДМИНИСТРАТОРА

Выполняя следующие действия на машине с включенным UAC-контролем, можно заставить Explorer запустить процесс либо с маркером обычного пользователя, либо с маркером администратора:

1. Войдите в систему под учетной записью, входящей в группу «Администраторы».
2. Щелкните на пунктах меню Пуск (Start), Все программы (Programs), Стандартные (Accessories), выберите пункт Командная строка (Command Prompt), щелкните правой кнопкой мыши на его значке и выберите пункт Запуск от имени Администратора (Run As Administrator). Вы увидите окно командной строки со словом Администратор (Administrator) в заголовке окна.
3. Повторите процесс, но теперь просто щелкните на значке, в результате будет запущено второе окно командной строки без привилегий администратора.
4. Запустите утилиту Process Explorer и просмотрите для двух запущенных вами процессов окон командной строки содержимое вкладок Security (Безопасность) диалоговых окно Properties (Свойства). Обратите внимание на то, что маркер обычного пользователя содержит SID только для запрета (deny-only) и мандатную метку Medium Mandatory Lavel и у него совсем немного привилегий. Свойства, показанные в правой части копии экрана, относятся к окну командной строки, запущенному с маркером администратора, а свойства в левой части относятся к окну командной строки, запущенному с фильтрованным маркером администратора.



Виртуальные учетные записи служб

Windows предоставляет специализированный тип учетной записи, известный как виртуальная учетная запись службы (или просто виртуальная учетная запись), для улучшения защитной изоляции и управления доступом служб Windows с минимальными административными усилиями (см. главу 4). Без этого механизма

службам Windows пришлось бы запускаться либо под одной из учетных записей, определенных Windows для ее встроенных служб (например, Local Service или Network Service), любо под обычной учетной записью домена. Такие учетные записи, как Local Service, совместно используются многими существующими службами и поэтому предоставляют ограниченную детализацию привилегий и ограниченное управление доступом, более того, они не могут управляться в доменном пространстве. Доменные учетные записи требуют периодического изменения пароля для соблюдения мер безопасности, и в ходе цикла смены пароля может быть затронута и доступность служб. Кроме того, для более качественной изоляции каждая служба должна выполняться под своей собственной учетной записью, но при использовании обычных учетных записей это в разы увеличивает усилия, затрачиваемые на управление.

При использовании виртуальных учетных записей служб каждая служба запускается под своей собственной учетной записью со своим собственным идентификатором безопасности. Имя учетной записи всегда начинается со строкового значения «NT SERVICE\», за которым следует внутренне имя службы. Виртуальные учетные записи служб могут появляться в списках управления доступом и могут быть связаны с привилегиями через групповую политику (Group Policy), как и всякое другое имя учетной записи. Но они не могут быть созданы или удалены с использованием обычных инструментов управления, а также не могут быть отнесены к каким-либо группам.

Windows автоматически устанавливает и периодически изменяет пароль виртуальной учетной записи службы. Подобно учетной записи «Локальная система и другие служебные учетные записи» (Local System and other service accounts), это действительно пароль, но он неизвестен системным администраторам.

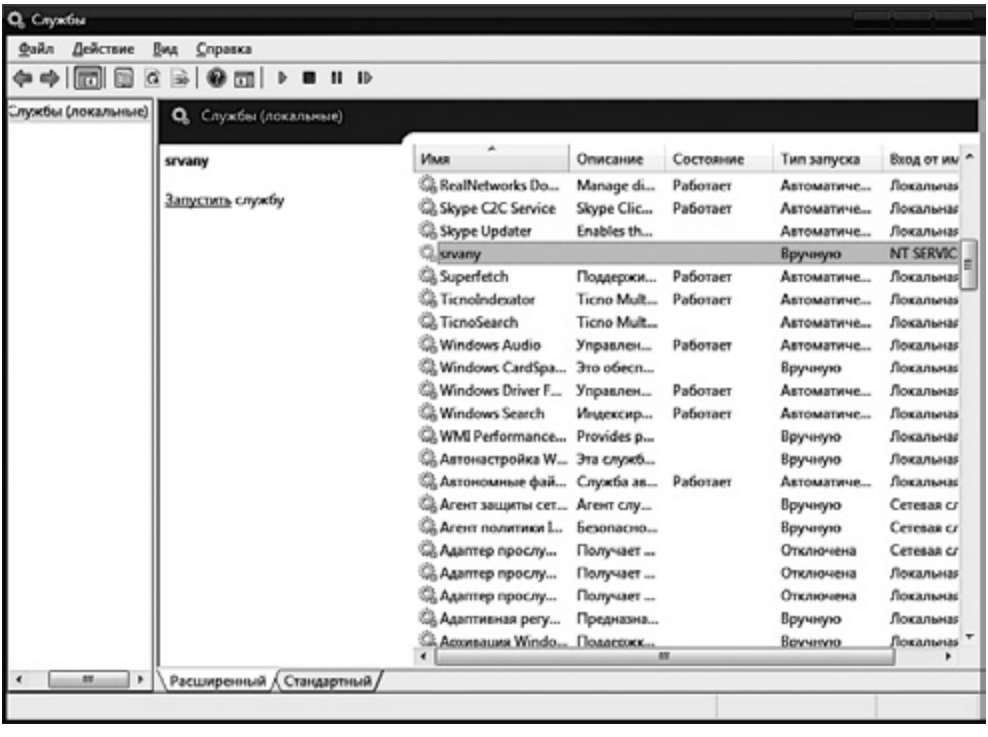
ЭКСПЕРИМЕНТ: ИСПОЛЬЗОВАНИЕ ВИРТУАЛЬНЫХ УЧЕТНЫХ ЗАПИСЕЙ СЛУЖБ

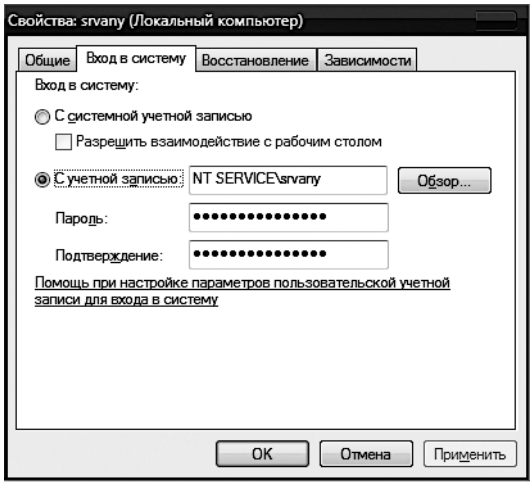
Создать службу, работающую под виртуальной учетной записью службы, можно с помощью средства Sc (service control, управление службой), выполнив следующие действия:

1. В окне командной строки, запущенном с правами администратора, воспользуйтесь командой создания create, указанной в утилите командной строки Sc (service control) для создания службы и той виртуальной учетной записи, под которой она будет запущена. В этом примере используется служба «srvany» из ранее используемого набора Windows Resource Kit:

```
C:\Windows\system32>sc create srvany obj= "NT SERVICE\srvany" binPath= "d:\a\
test\srvany.exe"
[SC] CreateService: успех
```

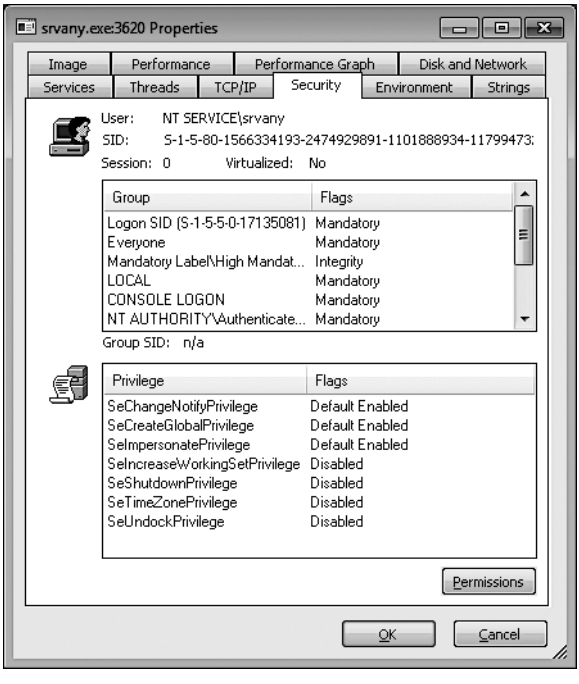
2. С помощью предыдущей команды создана служба (в реестре, а также во внутреннем списке диспетчера контроллера служб) и также создана виртуальная учетная запись службы. Теперь запустите MMC-оснастку Службы (Services) (services.msc), выберите новую службу и посмотрите на содержимое вкладки Вход в систему (Log On) в диалоговом окне Свойства (Properties).





3. Диалоговое окно свойств службы можно также использовать для создания виртуальной учетной записи для уже существующей службы. Для этого измените имя учетной записи на «NT SERVICE*имя_службы*» и очистите оба поля пароля. Но имейте в виду, что существующие службы под виртуальной учетной записью могут должным образом не запуститься, поскольку под этой учетной записью может не быть доступа к файлам или другим, необходимым службе ресурсам.

4. Если запустить Process Explorer и посмотреть на содержимое вкладки Security (Безопасность) диалогового окна Properties (Свойства), открытого для службы, использующей виртуальную учетную запись, то можно увидеть имя виртуальной учетной записи и ее идентификатор безопасности (SID).

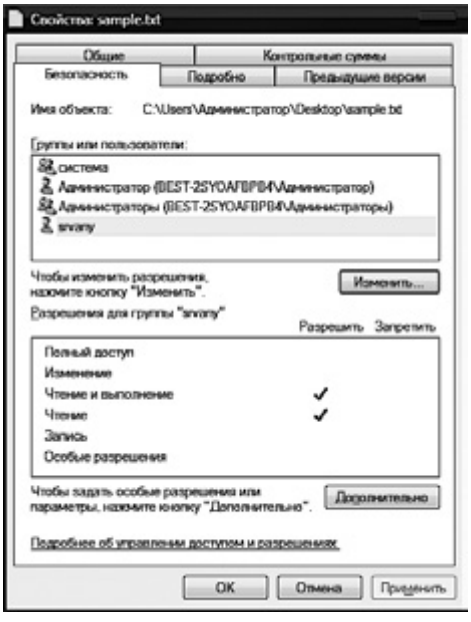


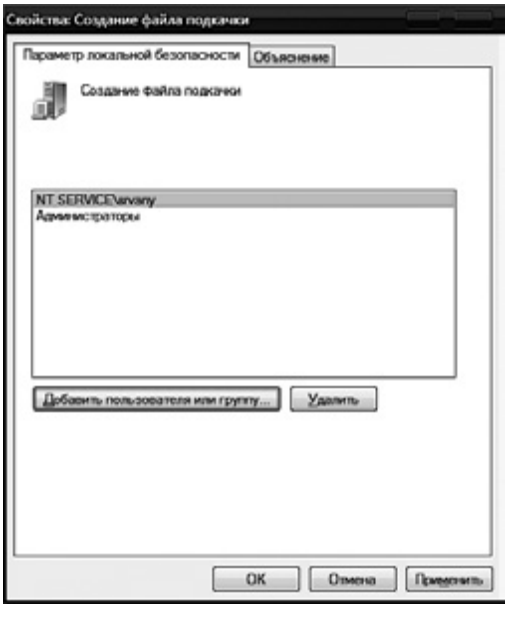
5. Виртуальная учетная запись службы может появиться в элементе управления доступом для любого объекта (например, файла), к которому служба должна иметь доступ. Если для файла открыть вкладку Безопасность (Security) в диалоговом окне Свойства (Properties) и создать ACL, ссылающийся на виртуальную учетную запись службы, вы увидите, что имя введенной вам учетной записи (например, NT SERVICE\srvany) было заменено просто именем службы (srvany) с помощью функции проверки имен, и оно появилось в списке управления доступом в укороченной форме.

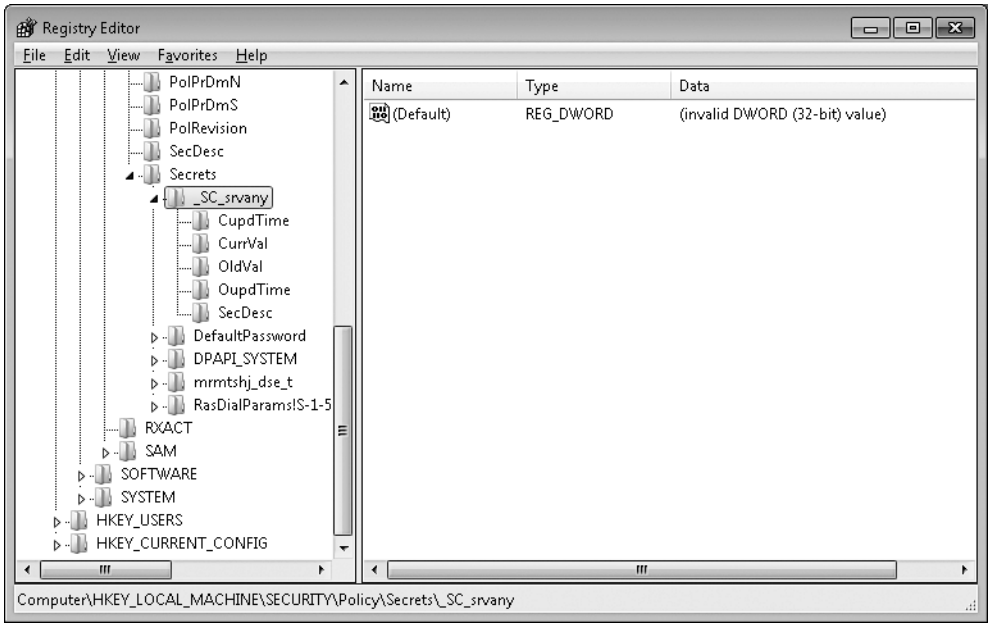
6. Разрешения (или права пользователя) виртуальной учетной записи службы могут быть предоставлены через групповую политику. В данном примере виртуальной учетной записи службы srvany предоставлено право создания файла подкачки.

7. А в пользовательских средствах администрирования, таких как lusrmgr.msc, виртуальные учетные записи служб видны не будут, потому что они в улье реестра SAM не сохраняются. Но если исследовать реестр в контексте встроенной учетной записи System (как описывалось ранее), вы увидите признаки учетной записи в разделе HKLM\Security\Policy\Secrets:

```
C:\>psexec -s -i -d c:\windows\regedit.exe
```







Дескрипторы безопасности и управление доступом

Маркеры, идентифицирующие учетные данные пользователя, являются только лишь частью уравнения, определяющего безопасность объекта. Другой частью уравнения является информация безопасности, связанная с объектом, которая определяет, кто и какие действия с объектом может выполнять. Структура дан-

ных для этой информации называется дескриптором безопасности. В дескриптор безопасности входят следующие атрибуты:

- **Номер версии.** Версия модели безопасности SRM, использованная для создания дескриптора.
- **Флаги.** Дополнительные модификаторы, определяющие поведение или характеристики дескриптора. Эти флаги перечислены в табл. 6.5.
- **SID владельца.** Идентификатор безопасности владельца.
- **SID группы.** Идентификатор безопасности основной группы объекта (используется только подсистемой POSIX).
- **Избирательный список управления доступом — Discretionary access control list (DACL).** Указывает, кто и какой доступ имеет к объекту.
- **Системный список управления доступом — System access control list (SACL).** Указывает, какие операции какими пользователями должны регистрироваться в журнале аудита безопасности и конкретный уровень целостности объекта.

Таблица 6.5. Флаги дескриптора безопасности

Флаг	Значение
SE_OWNER_ DEFAULTED	Обозначает дескриптор безопасности с идентификатором безопасности (SID) владельца, используемым по умолчанию. Этот бит используется для поиска всех объектов, имеющих установленный по умолчанию набор полномочий владельца
SE_GROUP_ DEFAULTED	Обозначает дескриптор безопасности с идентификатором безопасности (SID) группы, используемым по умолчанию. Этот бит используется для поиска всех объектов, имеющих установленный по умолчанию набор полномочий группы
SE_DACL_ PRESENT	Обозначает дескриптор безопасности, имеющий DACL. Если этот флаг не установлен или если этот флаг установлен, а DACL имеет значение NULL, дескриптор безопасности разрешает полный доступ для всех
SE_DACL_ DEFAULTED	Обозначает дескриптор безопасности с используемым по умолчанию DACL. Например, если создатель объекта не указал DACL, объект получает DACL, используемый по умолчанию, от маркера доступа создателя. Этот флаг может влиять на то, как система обрабатывает DACL с учетом наследования элемента управления доступом — access control entry (ACE). Система игнорирует этот флаг, если не установлен флаг SE_DACL_PRESENT
SE_SACL_ PRESENT	Обозначает дескриптор безопасности, у которого есть системный список управления доступом — system access control list (SACL)
SE_SACL_ DEFAULTED	Обозначает дескриптор безопасности с используемым по умолчанию SACL. Например, если создатель объекта не указал SACL, объект получает SACL по умолчанию от маркера доступа создателя. Этот флаг может повлиять на то, как система обрабатывает SACL с учетом наследования ACE. Система игнорирует этот флаг, если не установлен флаг SE_SACL_PRESENT

Флаг	Значение
SE_DACL_ UNTRUSTED	Обозначает, что ACL, указанный с помощью DACL дескриптора безопасности, был предоставлен ненадежным источником. Если этот флаг установлен и обнаружен составной ACE, система будет подставлять известные ей надежные SID-идентификаторы для серверных SID в ACE-элементах
SE_SERVER_ SECURITY	Требует, чтобы поставщик объекта, защищенный дескриптором безопасности, был сервером ACL, основанным на входящем ACL, безотносительно его источника (явно указанного или взятого по умолчанию). Это делается путем замены всех GRANT ACE-элементов составными ACE-элементами, предоставляющими текущий серверный доступ. Этот флаг имеет смысл только при заимствовании прав
SE_DACL_ AUTO_ INHERIT_REQ	Требует, чтобы поставщик объекта, защищенный дескриптором безопасности, автоматически распространял DACL на существующие дочерние объекты. Если поставщик поддерживает автоматическое наследование, DACL распространяется на любой существующий дочерний объект, а в дескрипторах безопасности родительского и дочернего объектов устанавливается бит SE_DACL_AUTO_INHERITED
SE_SACL_ AUTO_ INHERIT_REQ	Требует, чтобы поставщик объекта, защищенный дескриптором безопасности, автоматически распространял SACL на существующие дочерние объекты. Если поставщик поддерживает автоматическое наследование, SACL распространяется на любой существующий дочерний объект, а в дескрипторах безопасности родительского и дочернего объектов устанавливается бит SE_SACL_AUTO_INHERITED
SE_DACL_ AUTO_ INHERITED	Обозначает дескриптор безопасности, в котором DACL настроен на поддержку автоматического распространения наследования ACE-элементов на существующие дочерние объекты. Система устанавливает этот бит при выполнении алгоритма автоматического наследования для объекта и его существующих дочерних объектов
SE_SACL_ AUTO_ INHERITED	Обозначает дескриптор безопасности, в котором SACL настроен на поддержку автоматического распространения наследования ACE-элементов на существующие дочерние объекты. Система устанавливает этот бит при выполнении алгоритма автоматического наследования для объекта и его существующих дочерних объектов
SE_DACL_ PROTECTED	Предотвращает модификацию DACL дескриптора безопасности наследуемыми ACE-элементами.
SE_SACL_ PROTECTED	Предотвращает модификацию SACL дескриптора безопасности наследуемыми ACE-элементами
SE_RM_ CONTROL_ VALID	Обозначает надежность битов диспетчера управления ресурсом, имеющихся в дескрипторе безопасности. Под биты диспетчера управления ресурсом в структуре дескриптора безопасности отводится 8 разрядов, в которых содержится информация, относящаяся к доступу к структуре диспетчера управления ресурсом
SE_SELF_ RELATIVE	Обозначает дескриптор безопасности в самоопределяющемся относительном формате (self-relative format), при котором вся информация безопасности хранится в непрерывном блоке памяти. Если этот флаг не установлен, дескриптор безопасности находится в абсолютном формате

Список управления доступом (ACL) состоит из заголовка и нескольких элементов управления доступом (ACE), которых может и не быть. Существует два типа ACL-списков: DACL и SACL. В DACL в каждом ACE-элементе содержится SID и маска доступа (а также набор флагов, которые будут вскоре рассмотрены), которая обычно определяет права доступа (чтение, запись, удаление и т. д.), предоставленные или запрещенные держателю SID. Существует девять типов ACE-элементов, которые могут появляться в DACL: доступ разрешен, доступ запрещен, объект разрешен, объект запрещен, обратный вызов разрешен, обратный вызов запрещен, обратный вызов объекта разрешен, обратный вызов объекта запрещен и условные требования. Как вы, возможно, и ожидали, разрешающие доступ ACE предоставляют доступ пользователю, а запрещающие доступ ACE запрещают права доступа, указанные в маске доступа. ACE-элементы обратного вызова используются приложениями, применяющими API-интерфейс AuthZ (рассматриваемый чуть позже) для регистрации обратного вызова, который будет вызван AuthZ при выполнении проверки доступа с использованием этого ACE.

Разница между разрешенным объектом и разрешенным доступом и между запрещенным объектом и запрещенным доступом состоит в том, что типы объектов используются только в Active Directory. ACE-элементы этих типов имеют поле глобального уникального идентификатора — GUID, которое обозначает, что ACE применяется только к конкретным объектам или подчиненным объектам (к тем, у которых имеются GUID-идентификаторы). Кроме того, другой дополнительный GUID (128-разрядный идентификатор, гарантирующий его уникальность) обозначает тот тип дочернего объекта, который унаследует ACE при создании дочернего объекта в контейнере Active Directory, с примененным к нему ACE. ACE условных требований хранится в ACE-структуре типа *-callback и рассматривается в разделе, посвященном API-функциям AuthZ.

Аккумулирование прав доступа, предоставленных отдельными ACE-элементами, формирует набор прав доступа, предоставленных ACL-списком. Если в дескрипторе безопасности отсутствует DACL (то есть имеется нулевой DACL), полный доступ к объекту предоставляется кому угодно. Если DACL пуст (то есть в нем нуль ACE-элементов), никто не имеет доступа к объекту.

У ACE-элементов, используемых в DACL-списках, имеется набор флагов, управляющих характеристиками и определяющих характеристики ACE, связанные с наследованием. В некоторых пространствах имен объектов имеются контейнеры и объекты. Контейнер может содержать другие объекты-контейнеры и объекты-листья, являющиеся его дочерними объектами. Примерами контейнеров являются каталоги в пространстве имен файловой системы и разделы в пространстве имен реестра. Конкретные флаги в ACE управляют тем, как ACE распространяется на дочерние объекты контейнера, связанные с ACE. В табл. 6.6, частично повторяющей Windows SDK, перечислены правила наследования для флагов ACE.

В SACL содержатся два типа ACE-элементов, ACE-элементы системного аудита и ACE-элементы объекта системного аудита. Эти ACE-элементы определяют, какие операции выполняются в отношении объекта путем указания подвергаемых аудиту пользователей или групп.

Таблица 6.6. Правила наследования для флагов ACE

Флаг	Правило наследования
CONTAINER_ INHERIT_ACE	Дочерние объекты, являющиеся контейнерами, например каталоги, наследуют ACE в качестве действующего ACE-элемента. Унаследованный ACE является наследуемым до тех пор, пока не будет установлен битовый флаг NO_PROPAGATE_INHERIT_ACE
INHERIT_ONLY_ ACE	Этот флаг обозначает только наследуемый ACE, который не управляет доступом к объекту, к которому он прикреплен. Если этот флаг не установлен, ACE управляет доступом к тому объекту, к которому он прикреплен
INHERITED_ACE	Этот флаг обозначает, что ACE был унаследован. Система устанавливает этот бит при распространении наследуемого ACE на дочерний объект
NO_ PROPAGATE_ INHERIT_ACE	Если ACE унаследован дочерним объектом, система снимает флаги OBJECT_INHERIT_ACE и CONTAINER_INHERIT_ACE в унаследованном ACE. Это действие препятствует наследованию ACE следующими поколениями объектов
OBJECT_ INHERIT_ACE	Не являющиеся контейнером дочерние объекты наследуют ACE в качестве действующего ACE. Для дочерних объектов, являющихся контейнерами, ACE наследуется в качестве только наследуемого ACE, если только не установлен флаг NO_PROPAGATE_INHERIT_ACE

Информация об аудите хранится в системном журнале аудита — Audit Log. Как удачные, так и неудачные попытки могут быть подвержены аудиту. Как и в имеющихся в DACL двоюродных ACE, сориентированных на объекты, их системные родственники, ACE-элементы, сориентированные на объекты, указывают GUID, обозначающий типы объектов или подчиненных объектов, к которым применяется ACE, и дополнительный GUID, управляющий распространением ACE на конкретные типы дочерних объектов. Если SACL-список является нулевым, аудит в отношении объекта не проводится. Флаги наследования, применяемые к ACE-элементам DACL, также применяются к ACE-элементам системного аудита и к ACE-элементам объектов системного аудита.

На рис. 6.5 представлена упрощенная схема файлового объекта и его DACL.

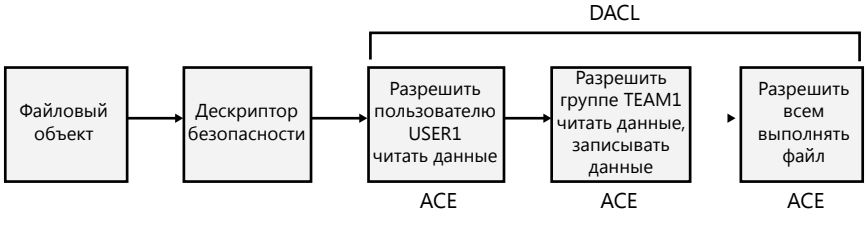


Рис. 6.5. Избирательный список управления доступом (DACL)

Как показано на рис. 6.5, первый ACE разрешает пользователю USER1 запрашивать файл. Второй ACE позволяет участникам группы TEAM1 иметь доступ к файлу по чтению и записи, а третий ACE предоставляет всем пользователям (Everyone) доступ к выполнению файла.

ЭКСПЕРИМЕНТ: ПРОСМОТР ДЕСКРИПТОРА БЕЗОПАСНОСТИ

Большинство подсистем исполняющей системы полагаются при управлении дескрипторами безопасности своих объектов на исходную функциональность безопасности диспетчера объектов. Исходные функции безопасности диспетчера объектов используют указатель дескриптора безопасности для хранения дескрипторов безопасности таких объектов. Например, диспетчер процессов использует исходные настройки безопасности, поэтому диспетчер объектов сохраняет дескрипторы безопасности процесса и потока в заголовках объектов процесса и потока соответственно. Указатель дескриптора безопасности событий, мьютексов и семафоров также хранит их дескрипторы безопасности. Для просмотра дескрипторов безопасности таких объектов при нахождении их заголовков можно воспользоваться работающей отладкой ядра, выполняя указанные далее действия. (Следует заметить, что и Process Explorer и AccessChk также могут показать дескрипторы безопасности для процессов.)

1. Запустите отладчик ядра.
2. Наберите команду !process 0 0 explorer.exe для получения информации о процессе Explorer:

```
lkd> !process 0 0 explorer.exe
PROCESS 85a3e030  SessionId: 1  Cid: 0aa4    Peb: 7ffd4000  ParentCid: 0a84
    DirBase: 0f419000  ObjectTable: 952cdd18  HandleCount: 1046.
    Image: explorer.exe
```

3. Наберите команду !object с адресом, который следует за словом PROCESS в выводе предыдущей команды в качестве аргумента для вывода структуры данных объекта:

```
lkd> !object 85a3e030
Object: 85a3e030  Type: (842339e0) Process
    ObjectHeader: 85a3e018 (new version)
    HandleCount: 8  PointerCount: 497
```

4. Наберите команду dt _OBJECT_HEADER и адрес поля заголовка объекта из вывода предыдущей команды, чтобы показать структуру данных заголовка объекта, включая значение указателя дескриптора безопасности:

```
lkd> dt _OBJECT_HEADER 85a3e018
nt!_OBJECT_HEADER
   +0x000 PointerCount       : 0n497
   +0x004 HandleCount        : 0n8
   +0x004 NextToFree         : 0x00000008 Void
   +0x008 Lock               : _EX_PUSH_LOCK
   +0x00c TypeIndex          : 0x7 ''
   +0x00d TraceFlags         : 0 ''
   +0x00e InfoMask           : 0x8 ''
   +0x00f Flags              : 0 ''
   +0x010 ObjectCreateInfo   : 0x8577e940 _OBJECT_CREATE_INFORMATION
   +0x010 QuotaBlockCharged  : 0x8577e940 Void
   +0x014 SecurityDescriptor : 0x97ed0b94 Void
   +0x018 Body               : _QUAD
```

5. И наконец, воспользуйтесь командой отладчика !sd для вывода дампа дескриптора безопасности. Указатель дескриптора безопасности в заголовке объекта использует некоторые младшие разряды в качестве флагов, которые должны быть обнулены до следующего указателя. На 32-разрядных системах имеется три бита флагов, поэтому в следующем примере с адресом дескриптора безопасности, показанном в структуре заголовка объекта, используется аргумент & −8. На 64-разрядных системах имеется четыре разряда флагов, поэтому вместо предыдущего аргумента нужно использовать аргумент & −10.

```
lkd> !sd 0x97ed0b94 & -8
->Revision: 0x1
->Sbz1    : 0x0
->Control : 0x8814
            SE_DACL_PRESENT
            SE_SACL_PRESENT
            SE_SACL_AUTO_INHERITED
            SE_SELF_RELATIVE
->Owner   : S-1-5-21-1488595123-1430011218-1163345924-1000
->Group   : S-1-5-21-1488595123-1430011218-1163345924-513
->Dacl    :
->Dacl    : ->AclRevision: 0x2
->Dacl    : ->Sbz1       : 0x0
->Dacl    : ->AclSize    : 0x5c
->Dacl    : ->AceCount   : 0x3
->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 : 0x001fffff
->Dacl    : ->Ace[0]: ->SID: S-1-5-21-1488595123-1430011218-1163345924-1000
->Dacl    : ->Ace[1]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[1]: ->AceFlags: 0x0
->Dacl    : ->Ace[1]: ->AceSize: 0x14
->Dacl    : ->Ace[1]: ->Mask : 0x001fffff
->Dacl    : ->Ace[1]: ->SID: S-1-5-18
->Dacl    : ->Ace[2]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[2]: ->AceFlags: 0x0
->Dacl    : ->Ace[2]: ->AceSize: 0x1c
->Dacl    : ->Ace[2]: ->Mask : 0x00121411
->Dacl    : ->Ace[2]: ->SID: S-1-5-5-0-178173
->Sacl    :
->Sacl    : ->AclRevision: 0x2
->Sacl    : ->Sbz1       : 0x0
->Sacl    : ->AclSize    : 0x1c
->Sacl    : ->AceCount   : 0x1
->Sacl    : ->Sbz2       : 0x0
->Sacl    : ->Ace[0]: ->AceType: SYSTEM_MANDATORY_LABEL_ACE_TYPE
->Sacl    : ->Ace[0]: ->AceFlags: 0x0
->Sacl    : ->Ace[0]: ->AceSize: 0x14
->Sacl    : ->Ace[0]: ->Mask : 0x00000003
->Sacl    : ->Ace[0]: ->SID: S-1-16-8192
```

Дескриптор безопасности содержит три разрешающие доступ ACE-элемента: один для текущего пользователя (S-1-5-21-1488595123-1430011218-1163345924-1000), один для учетной записи System (S-1-5-18) и последний для SID-входа в систему — Logon SID (S-1-5-5-0-178173). Список управления системным доступом имеет один элемент (S-1-16-8192), помечающий процесс как имеющий уровень целостности medium (средний). ■

Назначение ACL

Чтобы определить, какой DACL нужно назначить новому объекту, система безопасности использует первое подходящее правило из следующих четырех правил назначения:

1. Если при создании объекта вызывающий процесс предоставляет дескриптор безопасности явным образом, система безопасности применяет его к объекту. Если у объекта есть имя и он находится в объекте-контейнере (например, именованный объект-событие в каталоге \BaseNamedObjects пространства имен диспетчера объектов), система объединяет любые наследуемые ACE-элементы (ACE-элементы, которые могут распространяться из контейнера объекта) в DACL, если только у дескриптора безопасности не установлен флаг SE_DACL_PROTECTED, мешающий наследованию.
2. Если вызывающий процесс не предоставляет дескриптор безопасности и у объекта есть имя, система безопасности смотрит на дескриптор безопасности контейнера, в котором хранится имя нового объекта. Некоторые ACE-элементы каталога объекта могут быть помечены наследуемыми, следовательно, они должны быть применены к новым объектам, созданным в каталоге объекта. Если имеются какие-нибудь из этих наследуемых ACE-элементов, система безопасности формирует из них ACL-список, который прикрепляется к новому объекту. (Отдельные флаги обозначают ACE-элементы, которые должны быть унаследованы только объектами-контейнерами, а не объектами, не являющимися контейнерами.)
3. Если дескриптор безопасности не указан и объект не наследует никаких ACE-элементов, система безопасности извлекает исходный DACL из маркера доступа вызывающего процесса и применяет его к новому объекту. Некоторые подсистемы в Windows имеют жестко заданные DACL-списки, которые они присваивают создаваемому объекту (например, объектам служб, LSA и SAM).
4. Если дескриптор не указан, наследуемые ACE-элементы отсутствуют и нет исходного DACL-списка, система создает объект без DACL, что позволяет всем (всем группам и пользователям) иметь полный доступ к объекту.

Это правило похоже на третье правило, в котором маркер содержит нулевой исходный DACL. Правила, используемые системой при назначении новому объекту списка SACL, аналогичны тем правилам, которые применяются для назначения списка DACL, за некоторыми исключениями. Первое из них состоит в том, что наследуемые ACE-элементы системного аудита не распространяются на объекты, имеющие дескрипторы безопасности, помеченные флагом SE_SACL_PROTECTED (который похож на флаг SE_DACL_PROTECTED, защищающий списки DACL). Второе исключение состоит в том, что здесь нет указанных ACE-элементов системного аудита и нет наследуемых списков SACL, никакие SACL

к объекту не применяются. Это поведение отличается от того, которое применялось к исходным спискам DACL, поскольку у маркеров нет исходных SACL.

Когда новый дескриптор безопасности, содержащий наследуемые ACE-элементы, применяется к контейнеру, система автоматически распространяет наследуемые ACE-элементы на дескрипторы безопасности дочерних объектов. (Следует иметь в виду, что список DACL дескриптора безопасности не принимает наследуемые ACE-элементы DACL, если установлен его флаг SE_DACL_PROTECTED, и его SACL не наследует ACE-элементы SACL, если у дескриптора установлен флаг SE_SACL_PROTECTED.) Порядок, в котором наследуемые ACE-элементы объединяются с существующим дескриптором безопасности дочернего объекта таков, что любые ACE-элементы, которые были явным образом применены к ACL, хранятся перед теми ACE-элементами, которые были этим объектом унаследованы. Для распространения наследуемых ACE-элементов система применяет следующие правила:

- Если дочерний объект, не имеющий DACL, наследует ACE, это приводит к тому, что у этого дочернего объекта появляется DACL, содержащий только унаследованный ACE.
- Если дочерний объект с пустым DACL наследует ACE, это приводит к тому, что DACL у этого дочернего объекта содержит только унаследованный ACE.
- Применительно только к объектам Active Directory, если наследуемый ACE удаляется из родительского объекта, автоматически система наследования удаляет все копии ACE, унаследованные дочерними объектами.
- Применительно только к объектам Active Directory, если в результате автоматического наследования из DACL дочернего объекта удаляются все ACE-элементы, у дочернего элемента остается пустой, но все же имеющийся DACL.

Как вскоре обнаружится, порядок следования ACE-элементов в ACL является весьма важным аспектом модели безопасности Windows.

ПРИМЕЧАНИЕ

Вообще-то наследование для таких объектов хранения, как файловые системы, реестр или Active Directory, напрямую не поддерживается. API-интерфейсы Windows, поддерживающие наследование, включающие функцию SetEntriesInAcl, достигают своей цели путем вызова соответствующих функций в DLL-библиотеке поддержки наследования безопасности (%SystemRoot%\System32\Ntmarta.dll), которые знают, как проходить по таким объектам-хранилищам.

Определение возможности доступа

Для определения возможности доступа к объекту используются два метода:

- Обязательная проверка целостности, которая определяет, достаточно ли уровня целостности вызывающего процесса для доступа к ресурсу, на основе принадлежащего ресурсу уровня целостности и его мандатной политики.
- Проверка избирательного доступа, выявляющая, какой вид доступа к объекту имеется у конкретной учетной записи пользователя.

Когда процесс пытается открыть объект, проверка целостности осуществляется до стандартной Windows-проверки DACL в функции ядра SeAccessCheck,

поскольку она выполняется быстрее и может быстро исключить надобность осуществления полной проверки избирательного доступа. Согласно исходным политикам целостности в его маркере доступа (TOKEN_MANDATORY_NO_WRITE_UP и TOKEN_MANDATORY_NEW_PROCESS_MIN, рассмотренным ранее), процесс может открыть объект для доступа по записи, если его уровень целостности равен или выше уровня целостности объекта, и DACL также предоставляет процессу требуемый доступ. Например, процесс, имеющий низкий уровень целостности, не может открыть для записи объект, имеющий средний уровень целостности, даже если DACL предоставляет процессу право записи.

Применяя исходные политики целостности, процессы могут открыть любой объект, за исключением объектов процессов, потоков и маркеров, для доступа по чтению, если DACL этого объекта предоставляет им право на чтение. Это означает, что процесс, выполняемый с низким уровнем целостности, может открыть любые файлы, доступные пользовательской учетной записи, под которой он был запущен. Защищенный режим Internet Explorer — Protected Mode Internet Explorer — использует уровни целостности, чтобы помочь противодействовать вредоносным программам, ведущим заражение путем модификации настроек учетной записи пользователя, но он не мешает вредоносной программе читать документы пользователя.

Следует напомнить, что исключение составляют объекты процесса и потока, поскольку их политика целостности также включает запрет на чтение — No-Read-Up. Это означает, что уровень целостности процесса должен быть равен или выше уровня целостности того процесса или потока, который он хочет открыть, и что DACL должен предоставить ему требуемые доступы, чтобы попытка открытия завершилась успешно. Предполагая, что списки DACL разрешают требуемый доступ, на рис. 6.6 показаны типы доступа, имеющиеся у процесса, выполняемого со средним или низким уровнем целостности к другим процессам и объектам.

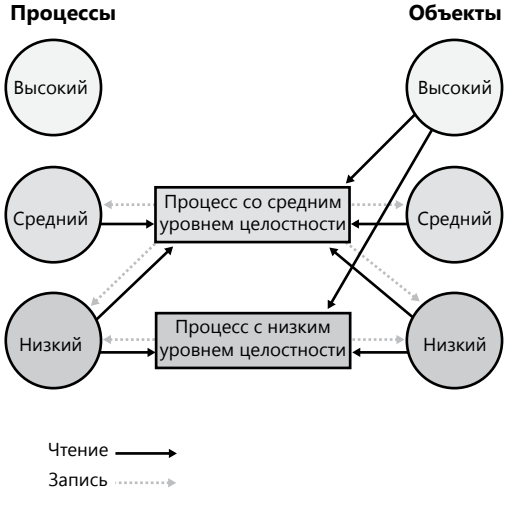


Рис. 6.6. Доступ процессов к объектам для процессов со средним и низким уровнем целостности

ИЗОЛЯЦИЯ ПРИВИЛЕГИЙ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА

Имеющаяся в Windows подсистема сообщений также соблюдает уровни целостности для реализации изоляции привилегий пользовательского интерфейса — User Interface Privilege Isolation (UIPI). Подсистема осуществляет эту изоляцию тем, что мешает процессу отправлять оконные сообщения окнам, которыми владеет процесс, имеющий более высокий уровень целостности, за исключением следующих информационных сообщений:

- WM_NULL
- WM_MOVE
- WM_SIZE
- WM_GETTEXT
- WM_GETTEXTLENGTH
- WM_GETHOTKEY
- WM_GETICON
- WM_RENDERFORMAT
- WM_DRAWCLIPBOARD
- WM_CHANGECBCHAIN
- WM_THEMECHANGED

Такое использование уровней целостности не дает процессам обычного пользователя вводить информацию в окна процессов с повышенными привилегиями или не дает проводить так называемые подрывные атаки — shatter attack (например, отправку процессу бесформенных сообщений, вызывающих переполнение внутреннего буфера, что может привести к выполнению кода на уровне процесса с повышенными привилегиями). UIPI также блокирует работу перехватчиков событий окон от имитации окон процессов с более высоким уровнем целостности, чтобы процесс обычного пользователя не мог регистрировать клавиатурные последовательности, которые, к примеру, набираются пользователем в административном приложении. Аналогичным образом блокируются также и перехватчики журналов, чтобы не позволить процессам с более низким уровнем целостности отслеживать поведение процессов с более высоким уровнем целостности.

Процессы могут выбрать разрешение дополнительных сообщений, чтобы пройти охрану путем вызова API-функции ChangeWindowMessageEx. Обычно эта функция используется для добавления сообщений, требуемых пользовательским элементам управления для обмена данными за пределами вне обычных общих элементов управления Windows. Более старая API-функция ChangeWindowMessageFilter выполняет те же действия, но в отношении процесса, а не окна. При использовании функции ChangeWindowMessageFilter два пользовательских элемента управления внутри одного и того же процесса могут быть использованы одними и теми же внутренними оконными сообщениями, что может привести к разрешению потенциально опасного оконного сообщения в одном из элементов управления, просто потому, что оно будет только лишь сообщением запроса для другого пользовательского элемента управления.

Поскольку под ограничения UIPI подпадают такие приложения для прощения работы с компьютером, как экранная клавиатура (Osk.exe) (что потребует от адаптационных приложений исполняться для каждой разновидности видимых процессов рабочего стола, имеющих уровни целостности), эти процессы могут включить доступ к пользовательскому интерфейсу (UI Access).

Этот флаг может присутствовать в файле манифеста, который принадлежит образу, и благодаря ему процесс будет запущен с немного более высоким уровнем целостности по сравнению со средним (между 0x2000 и 0x3000) под учетной записью обычного пользователя, или с высоким уровнем целостности, если запускается под учетной записью администратора. Следует иметь в виду, что во втором случае запрос на повышение полномочий не будет отображен. Чтобы процесс установил этот флаг, его образ должен быть подписан и находиться в одном из нескольких безопасных мест, включая %SystemRoot% и %ProgramFiles%.

После завершения проверки целостности и предположения, что мандатная политика разрешает доступ к объекту на основе уровня целостности вызывающего процесса, для проверки избирательного доступа к объекту используется один из двух алгоритмов, который определит окончательный результат проверки возможности доступа:

- Определение максимального разрешенного доступа к объекту, форма которого экспортируется в пользовательский режим с помощью Windows-функции GetEffectiveRightsFromAcl. Это определение также используется, когда программа указывает в качестве желаемого доступа MAXIMUM_ALLOWED, для устаревших API-функций, у которых не используется параметр желаемого доступа.
- Определение, разрешен ли указанный желаемый доступ, что может быть сделано с помощью Windows-функции AccessCheck или функции AccessCheckByType.

Первый алгоритм проводит следующую проверку элементов DACL:

1. Если у объекта нет DACL (нулевой DACL), объект не имеет защиты и система безопасности предоставляет полный доступ.
2. Если у вызывающего процесса есть привилегия получения объекта во владение (take-ownership), система безопасности предоставляет перед исследованием DACL доступ по записи в качестве владельца (write-owner). (Привилегии получения владения и доступа к записи в качестве владельца будут вскоре рассмотрены.)
3. Если вызывающий процесс является владельцем объекта, система ищет SID с правами владельца — OWNER_RIGHTS-и использует его в качестве SID для следующих действий. В противном случае предоставляются права управления чтением (read-control) и записи DACL (write-DACL).
4. Для каждого ACE-элемента запрещения доступа (access-denied), который содержит SID, совпадающий с SID-идентификатором маркера доступа вызывающего процесса, маска доступа ACE-элемента удаляется из маски предоставляемого доступа (granted-access mask).
5. Для каждого ACE-элемента разрешения доступа, который содержит SID, совпадающий с SID-идентификатором маркера доступа вызывающего процесса, маска доступа ACE-элемента добавляется к вычисленной маске предоставляемого доступа, если только этот доступ уже не был запрещен.

Когда будут проверены все элементы DACL, вычисленная маска предоставляемого доступа возвращается вызывающему процессу в виде максимально разрешенного доступа к объекту. Эта маска представляет собой совокупный набор

типов доступа, который вызывающий процесс может успешно запросить при открытии объекта.

Предыдущее описание применимо только к форме алгоритма режима ядра. Windows-версия, реализуемая функцией GetEffectiveRightsFromAcl, отличается тем, что она не выполняет действие 2 и рассматривает не маркер доступа, а SID отдельного пользователя или группы.

ДРУГИЕ ПРАВА

Поскольку владельцы объекта могут, как правило, отменить настройки безопасности объекта за счет безусловного получения прав на управление чтением и на запись DACL, специализированные методы управления этим поведением выражаются в Windows с помощью SID прав владельца — Owner Rights SID.

SID прав владельца создается по двум основным причинам: для улучшения работы служб в операционной системе и для достижения большей гибкости при определенных сценариях использования. Предположим, например, что администратор хочет разрешить пользователям создавать файлы и папки без модификации ACL-списков таких объектов. (Пользователи могут случайно или преднамеренно предоставить доступ к этим файлам или папкам с нежелательных учетных записей.) Используя наследуемый SID прав владельца, пользователям можно помешать редактировать или даже просматривать ACL создаваемых ими объектов. Второй сценарий использования относится к групповым изменениям. Предположим, что работник, который входит в некую группу особо посвященных или важных персон, создал несколько файлов, будучи участником этой группы, а теперь он удален из этой группы по каким-то деловым соображениям. Поскольку этот работник все еще остается пользователем, он может по-прежнему иметь доступ к важным файлам.

Как уже упоминалось, Windows также использует SID прав владельца для улучшения работы служб. Когда служба во время своего выполнения создает объект, SID владельца, связанный с этим объектом, является учетной записью, под которой запущена служба (например, учетной записью локальной системы или локальной службы), и не является SID самой службы. Это означает, что любая друга служба, работающая под той же учетной записью, будет иметь доступ к объекту, будучи его владельцем. SID прав владельца пресекает такое нежелательное поведение.

Второй алгоритм используется для определения, может ли быть предоставлен конкретный запрашиваемый доступ на основе маркера доступа вызывающего процесса. У каждой функции открытия в Windows API, работающей с защищенными объектами, имеется параметр, указывающий желаемую маску доступа, являющуюся последним компонентом уравнения безопасности. Чтобы определить, имеет ли вызывающий процесс доступ, выполняются следующие действия:

1. Если у объекта нет DACL (нулевой DACL), объект не имеет защиты и система безопасности предоставляет желаемый доступ.
2. Если у вызывающего процесса есть привилегия получения объекта во владение (take-ownership), система безопасности предоставляет перед исследованием DACL доступ по записи в качестве владельца (write-owner). Но если доступ по записи в качестве владельца был единственным запрошенным доступом со стороны вызывающего процесса, имеющего привилегии получения объекта во владение, система безопасности предоставляет этот доступ и не исследует DACL.

3. Если вызывающий процесс является владельцем объекта, система ищет SID с правами владельца — OWNER_RIGHTS — и использует его в качестве SID для следующих действий. В противном случае предоставляются права управления чтением (read-control) и записи DACL (write-DACL). Если эти права были единственными правами, запрошенными вызывающим процессом, доступ предоставляется без исследования DACL.
4. Исследуется каждый ACE-элемент DACL с первого до последнего. В случае удовлетворения одному из следующих условий ведется обработка ACE:
   1) ACE является запрещающим доступ, и SID в ACE совпадает с включенным SID (SID-идентификаторы могут быть включенными и выключенными) или с SID только для запрета в маркере доступа вызывающего процесса.
   2) ACE является разрешающим доступ, и SID в ACE совпадает с включенным SID в маркере доступа вызывающего процесса, который не относится к типу маркеров только для запрета.
   3) Это второй проход по дескриптору для проверки на наличие ограниченных SID, и SID в ACE совпадает с ограниченным SID в маркере доступа вызывающего процесса.
   4) ACE не помечен как предназначенный только для наследования.
   5) Если ACE является разрешающим доступ, то в маске доступа в ACE предоставляются запрошенные права доступа; если были предоставлены все запрошенные права, то проверка доступа достигает своей цели. Если ACE является запрещающим доступ и какое-либо из запрошенных прав доступа относится к запрещенным правам доступа, доступ к объекту запрещается.
   6) Если проверка дошла до конца DACL и некоторые из запрошенных прав доступа все еще не были предоставлены, доступ запрещается.
   7) Если предоставляются все доступы, но в маркере доступа вызывающего процесса есть хотя бы один ограниченный SID, система повторно сканирует ACE-элементы DACL, проводя поиск таких элементов, у которых маска доступа совпадает с запрошенными пользователем правами доступа, и поиск совпадения SID ACE-элемента с любым ограниченным SID вызывающего процесса. Доступ к объекту предоставляется пользователю только в том случае, если запрошенные права доступа предоставляются при обоих сканированиях DACL.

Поведение обоих алгоритмов проверки прав доступа зависит от относительного порядка следования разрешающих и запрещающих ACE-элементов. Рассмотрим объект, у которого имеются только два ACE-элемента, где один ACE указывает, что конкретному пользователю разрешен полный доступ к объекту, а другой ACE запрещает пользовательский доступ. Если разрешающий ACE предшествует запрещающему, пользователь может получить полный доступ к объекту, но при обратном порядке пользователь не может получить доступ к объекту.

Ряд функций Windows, например SetSecurityInfo и SetNamedSecurityInfo, применяют ACE-элементы в предпочитаемом порядке, где явно запрещающие ACE-элементы предшествуют явно разрешающим. Следует заметить, что эти функции используются диалоговыми окнами редактирования безопасности, с помощью которых вы, к примеру, редактируете права доступа к NTFS-файлам и разделам реестра. Функции SetSecurityInfo и SetNamedSecurityInfo также при-

меняют правила наследования ACE к дескриптору безопасности, в отношении которого они применяются.

На рис. 6.7 показан пример проверки доступа, показывающий важность порядка следования ACE. В этом примере запрещен доступ к желаемому пользователем открытию файла даже при том, что ACE-элемент в DACL объекта предоставляет доступ, поскольку ACE, запрещающий пользовательский доступ (в силу его принадлежности к группе Writers), предшествует ACE, предоставляющего доступ.

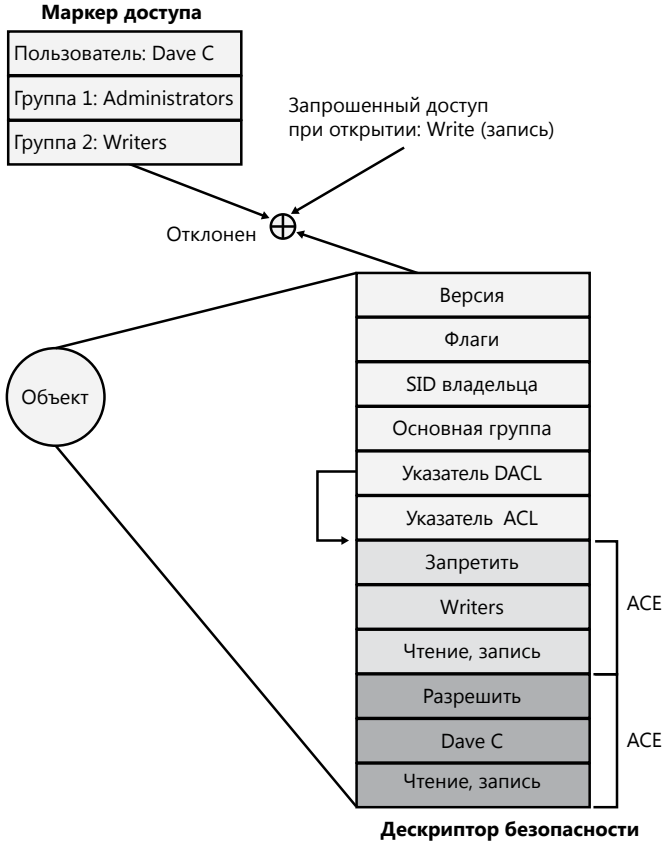


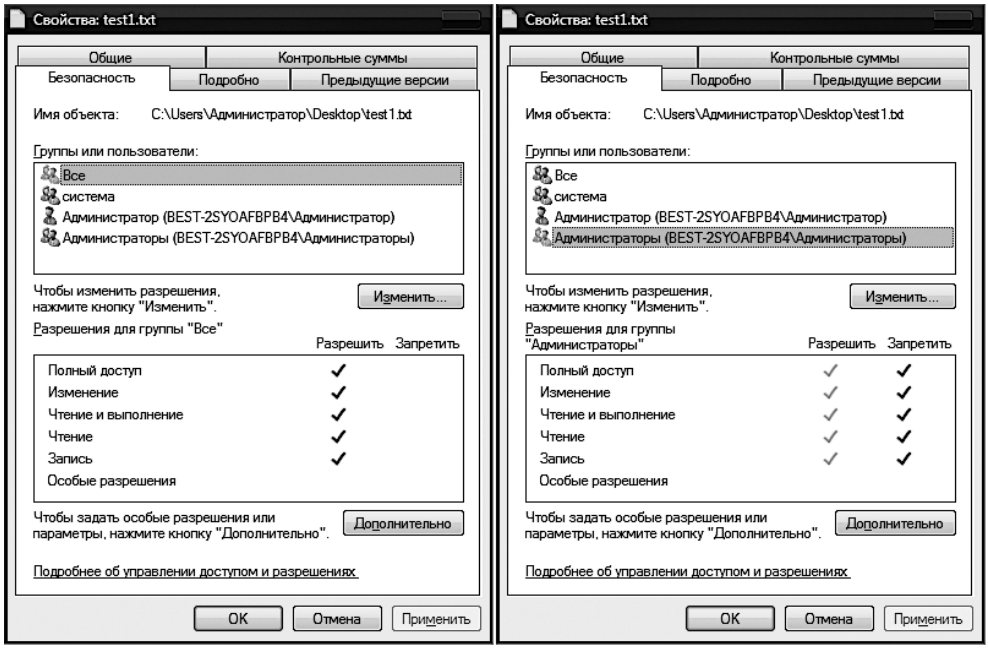
Рис. 6.7. Пример проверки прав доступа

Как уже было установлено, поскольку обрабатывать системе безопасности DACL при каждом использовании процессом дескриптора будет слишком неэффективно, SRM проводит эту проверку прав доступа только при открытии дескриптора, а не при каждом его использовании. Следовательно, если процесс успешно открывает дескриптор, система безопасности не может отменить предоставленные права доступа, даже при изменении DACL объекта. Также следует иметь в виду, что поскольку код режима ядра использует для доступа к объектам указатели, а не дескрипторы, при использовании объектов операционной системой проверка прав доступа не производится. Иными словами, исполняющая система Windows доверяет самой себе (и всем загруженным драйверам) в смысле безопасности.

Тот факт, что владельцу объекта всегда предоставляется доступ к записи DACL этого объекта, означает, что пользователи ни при каких условиях не могут быть отстранены от доступа к объектам, владельцами которых они являются. Если же по каким-то причинам у объекта есть пустой DACL (без доступа к нему), владелец все равно может открыть объект с доступом к DACL по записи, а затем применить к нему новый DACL с нужными правами доступа.

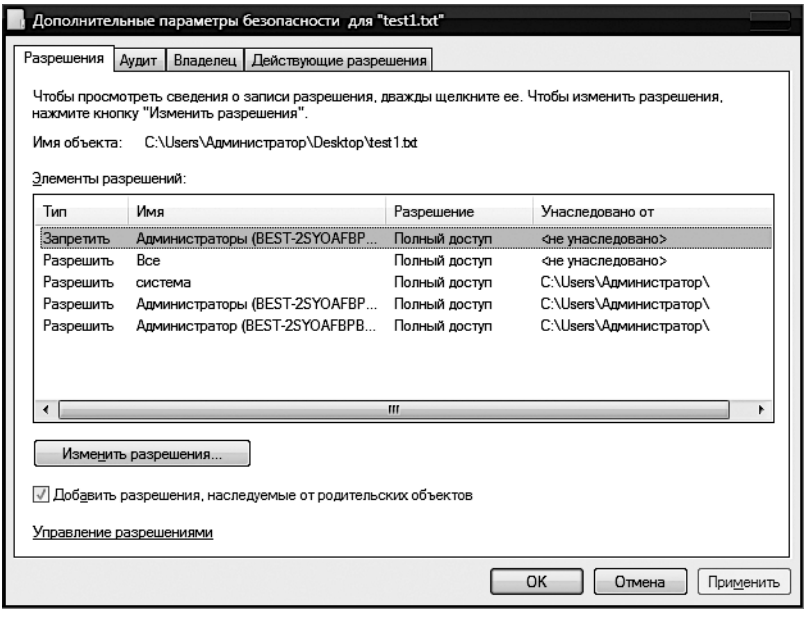
ПРЕДУПРЕЖДЕНИЕ, КАСАЮЩЕЕСЯ РЕДАКТОРОВ БЕЗОПАСНОСТИ С ГРАФИЧЕСКИМ ИНТЕРФЕЙСОМ

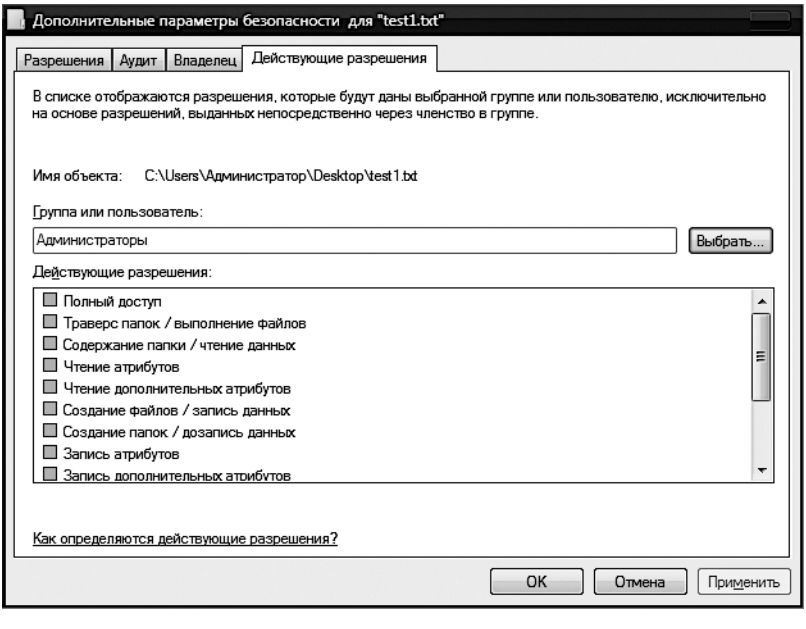
При использовании редакторов прав доступа с графическим интерфейсом для настройки файловых объектов, объектов реестра или Active Directory, или какого-нибудь другого защищаемого объекта главное диалоговое окно настроек безопасности показывает вам потенциально дезориентирующий вид таких настроек применительно к объекту. Если вы разрешите Полный доступ (Full Control) для группы Все (Everyone) и запретите его для группы Администраторы (Administrator), список может заверить вас, что разрешающий доступ ACE-элемент группы Все (Everyone) предшествует запрещающему доступ ACE-элементу группы Администраторы (Administrator), поскольку это порядок, в котором они появляются. Но, как уже было сказано, редакторы применительно к ACL объекта помещают запрещающие доступ ACE до разрешающих доступ ACE.



Вкладка Разрешения (Permissions) диалогового окна Дополнительные параметры безопасности (Advanced Security Settings) показывает порядок следования ACE-элементов в DACL. Но даже это диалоговое окно может внести путаницу, поскольку в составных DACL запрещающие доступ ACE-элементы для различных доступов могут следовать за разрешающими ACE-элементами для других типов доступа.

Единственный точный способ узнать о разрешениях доступов конкретного пользователя или группы к объекту (кроме того, чтобы попытаться получить от имени этого пользователя или группы доступ к объекту) заключается в использовании вкладки Действующие разрешения (Effective Permissions) диалогового окна, которая появляется после щелчка на кнопке Дополнительно (Advanced) в диалоговом окне Свойства (Properties). Введите имя проверяемого пользователя или группы, и в диалоговом окне будет показано, какие права доступа разрешены для них к объекту.





AuthZ API

Windows API-интерфейс AuthZ предоставляет функции авторизации и реализует такую же модель безопасности, что и монитор безопасности, но он реализует эту модель полностью в пользовательском режиме в библиотеке %SystemRoot%\System32\Authz.dll. Благодаря ему приложения, желающие защитить свои собственные закрытые объекты, например таблицы баз данных, получают возможность воспользоваться моделью безопасности Windows, не затрачиваясь на переходы между пользовательским режимом и режимом ядра, которые могли бы им понадобиться при расчете на монитор безопасности.

API-функции AuthZ используют стандартные структуры данных дескриптора безопасности, SID-идентификаторы и привилегии. Вместо использования маркеров для представления клиентов, AuthZ использует AUTHZ_CLIENT_CONTEXT. AuthZ включает эквиваленты пользовательского режима для всех имеющихся в Windows функций проверок безопасности доступа, например функцию AuthzAccessCheck в версии AuthZ функции AccessCheck из Windows API, которая использует функцию SeAccessCheck монитора безопасности.

Другим преимуществом, доступным приложениям, использующим AuthZ, является то, что они могут заставить AuthZ сохранять в кэш-памяти результаты проверок безопасности для упрощения последующих проверок, использующих тот же контекст и дескриптор безопасности клиента. Полная документация по AuthZ приведена в Windows SDK.

Рассмотренные ранее механизмы безопасности, относящие к управлению избирательным доступом, являлись частью семейства Windows NT с самого начала, и они достаточно хорошо работали в статической, контролируемой среде. Этот тип проверки доступа, использующий идентификаторы безопасности (SID) и участие в группах безопасности, известен как управление доступом, основанное на избирательном подходе — identity-based access control (IBAC), и он требует, чтобы система безопасности различала идентичность каждого возможного средства доступа, когда в дескриптор безопасности объекта помещается DACL.

Windows включает поддержку управления доступом на основе заявок — Claims Based Access Control (CBAC), — где доступ предоставляется не на основе идентичности или групповой принадлежности средства доступа, а на основе произвольных атрибутов, назначаемых средству доступа и сохраняемых в его маркере доступа. Атрибуты предоставляются поставщиком атрибутов, например AppLocker. Механизм CBAC предоставляет множество преимуществ, включая возможность создания DACL для пользователя, чья идентичность еще не известна, или динамически вычисляемые атрибуты пользователя.

CBAC ACE-элемент (известный так же как условный ACE) хранится в ACE-структуре *-callback, которая полностью относится к AuthZ и игнорируется системной API-функцией SeAccessCheck. Процедура режима ядра SeSrpAccessCheck не понимает условных ACE-элементов, поэтому использовать CBAC могут только приложения, вызывающие API-функции AuthZ. Единственным системным компонентом, использующим CBAC для установки таких атрибутов, как путь или издатель, является AppLocker. Приложения сторонних разработчиков могут использовать CBAC путем применения API-функций CBAC, имеющихся в интерфейсе AuthZ.

Использование проверок безопасности CBAC позволяет применять эффективные политики управления, к числу которых относятся:

- Запуск только тех приложений, которые утверждены IT-отделом.
- Разрешение только одобренным приложениям получать доступ к контактам вашего Microsoft Outlook или календаря.
- Разрешение только сотрудникам определенного этажа здания получать доступ к принтерам этого этажа.
- Разрешение доступа к веб-сайтам внутренней сети только штатным работникам (в отличие от подрядчиков).

На атрибуты можно ссылаться в так называемом условном ACE-элементе, где проверяется присутствие, отсутствие или значение одного или нескольких атрибутов. Имя атрибута может состоять из любых алфавитно-цифровых символов Unicode, а также из символов «:/._». В качестве значения могут использоваться: 64-разрядное целое число, строка Unicode, строка байтов или массив.

Условные ACE-элементы

Формат строк языка определения дескрипторов безопасности — SDDL (Security Descriptor Definition Language) был расширен для поддержки ACE-элементов с условными выражениями. Новый формат строки SDDL имеет следующий вид: AceType;AceFlags;Rights;ObjectGuid;InheritObjectGuid;AccountSid;(Conditional-Expression).

Типом ACE (AceType) для условного ACE является либо XA (для разрешающего SDDL_CALLBACK_ACCESS_ALLOWED), либо XD (для запрещающего SDDL_CALLBACK_ACCESS_DENIED). Следует заметить, что ACE-элементы с условными выражениями используются для авторизации по заявке (конкретно функциями AuthZ API и AppLocker) и не распознаются диспетчером объектов или файловыми системами.

Условное выражение может включать любые элементы, показанные в табл. 6.7.

Таблица 6.7. Допустимые элементы для условных выражений

Элемент выражения	Описание
Имя_атрибута	Проверяет наличие у конкретного атрибута наличие ненулевого значения
exists *Имя_атрибута*	Проверяет существование указанного атрибута в контексте клиента
Имя_атрибута оператор значение	Возвращает результат указанной операции. Для тестирования значений атрибутов для использования в условных выражениях используются следующие бинарные (в отличие от унарных) операторы, имеющие форму *Имя_атрибута оператор значение*: Contains any_of , ==, !=, <, <=, >, >=
Условное_выражение || *Условное_выражение*	Проверяет, не имеет ли любое из указанных условных выражений значение true
Условное_выражение && *Условное_выражение*	Проверяет, не имеют ли оба из указанных условных выражений значение true

Таблица 6.7 *(продолжение)*

Элемент выражения	Описание
!(*Условное_выражение*)	Превращает уловное выражение в его противоположность
Member_of{SidArray}	Проверяет, содержит ли массив SID_AND_ATTRIBUTES, относящийся к контексту клиента, все идентификаторы безопасности (SID) в списке с запятыми в качестве разделителей, определенным с помощью SidArray

Условный ACE-элемент может содержать любое количество условий, и он либо игнорируется, если в результате вычисления условия будет получено значение false, либо применяется, если результат равен true. Условный ACE может быть добавлен к объекту с использованием API-функции AddConditionalAce и проверен с помощью API-функции AuthzAccessCheck.

Условный ACE может указать, какой доступ к конкретным записям данных внутри программы должен быть предоставлен только тому пользователю, который отвечает следующим критериям:

- У него имеется атрибут Role (роль) со значением Architect (создатель), Program Manager (Диспетчер программ) или Development Lead (ведущий разработчик) и атрибут Division (подразделение) со значением Windows.
- Его атрибут ManagementChain (цепочка управления) содержит значение John Smith.
- Его атрибут CommissionType (тип полномочий) имеет значение Officer (служащий), а значение его атрибута PayGrade (разряд заработной платы) выше 6 (то есть соответствует званию генерала — General Officer в вооруженных силах США).

В составе Windows нет средств для просмотра и редактирования условных ACE-элементов.

Права доступа и привилегии

Многие операции, выполняемые процессами в ходе их работы, не могут быть авторизованы через защиту доступа к объекту, поскольку они не касаются взаимодействия с конкретным объектом. Например, возможность обхода проверок безопасности при открытии файлов для резервного копирования является свойством учетной записи, а не конкретного объекта. Чтобы разрешить системному администратору управлять тем, под какими учетными записями можно выполнять связанные с безопасностью операции, в Windows используются как привилегии, так и права доступа.

Привилегии являются правом выполнять под той или иной учетной записью конкретную, связанную с системой операцию, например выключение компьютера или изменение системного времени. Право учетной записи разрешает или запрещает той учетной записи, которой оно назначено, выполнять конкретный тип входа в систему, например локальный или интерактивный.

Системный администратор назначает привилегии группам и учетным записям, используя такие инструментальные средства, как MMC-оснастка «Пользователи и группы Active Directory» — Active Directory Users and Groups, — для домен-

ных учетных записей или редактор локальной политики безопасности — Local Security Policy Editor (%SystemRoot%\System32\secpol.msc). Доступ к этому редактору можно получить в папке Администрирование (Administrative Tools) Панели управления (Control Panel) или в меню Пуск (Start) (если это меню настроено на присутствие ссылки Администрирование). На рис. 6.8 показывается конфигурация Назначение прав пользователя (User Rights Assignment) в редакторе Локальная политика безопасности (Local Security Policy Editor), где выведен полный список привилегий и прав учетных записей, доступных на Windows. Следует заметить, что это средство не делает различий между приви-

Рис. 6.8. Назначение прав пользователей через редактор локальной политики безопасности

легиями и правами учетных записей. Но вы можете отличить их друг от друга, поскольку любые права пользователя, не содержащие слова «вход в», являются привилегиями учетных записей.

Права учетной записи

Права учетной записи не навязываются монитором безопасности и не хранятся в маркерах доступа. Функцией, отвечающей за вход в систему, является LsaLogonUser. Служба Winlogon, к примеру, вызывает API-функцию LogonUser при интерактивном входе пользователя на компьютер, а функция LogonUser вызывает функцию LsaLogonUser. Функции LogonUser передается параметр, включающий тип выполняемого входа в систему, который может быть интерактивным, сетевым, пакетным, служебным и в качестве клиента терминального сервера.

В ответ на запросы входа в систему механизм проверки подлинности локальной системы безопасности — Local Security Authority (LSA) — извлекает права учетной записи, назначенные пользователю, из базы данных политики LSA при попытке пользователя войти в систему. LSA сверяет тип входа в систему с правами учетной записи, назначенными той учетной записи, под которой происходит вход в систему, и отклоняет вход, если у учетной записи нет прав, разрешающих данный тип входа в систему. В табл. 6.8 перечислены права пользователя, определяемые системой Windows.

Таблица 6.8. Права учетной записи

Пользовательское право	Роль
Deny logon locally (запрет на локальный вход в систему), Allow logon locally (разрешение на локальный вход в систему)	Используется для интерактивных входов в систему, происходящих на локальной машине
Deny logon over the network (запрет на вход в систему по сети), Allow logon over the network (разрешение на вход в систему по сети)	Используется для входов в систему, происходящих на удаленной машине
Deny logon through Terminal Services (запрет на вход в систему через службу терминалов), Allow logon through Terminal Services (разрешение на вход в систему через службу терминалов)	Используется для входов в систему в качестве клиента службы терминалов
Deny logon as a service (запрет на вход в систему в качестве службы), Allow logon as a service (разрешение на вход в систему в качестве службы)	Используется диспетчером управления службами при запуске той или иной службы под конкретной учетной записью
Deny logon as a batch job (запрет на вход в систему в качестве пакетного задания), Allow logon as a batch job (разрешение на вход в систему в качестве пакетного задания)	Используется при выполнении входа в систему пакетного типа

Windows-приложения с помощью функций LsaAddAccountRights и LsaRemoveAccountRights могут добавить или удалить права пользователя из учетной записи, а с помощью функции LsaEnumerateAccountRights они могут определить, какие права назначены учетной записи.

Привилегии

Количество привилегий, определяемых операционной системой, со временем существенно увеличилось. В отличие от прав пользователя, которые задаются LSA в одном месте, различные привилегии определяются разными компонентами, которые их же и применяют. Например, привилегия отладки, позволяющая процессу обходить проверки безопасности при открытии дескриптора другого процесса с помощью API-функции Windows OpenProcess, проверяется диспетчером процесса. В табл. 6.9 представлен полный список привилегий и дано описание того, как и когда они проверяются системными компонентами.

Когда компоненту нужно проверить маркер с целью определения наличия привилегии, им используются API-функции PrivilegeCheck или LsaEnumerateAccountRights при работе в пользовательском режиме и SeSinglePrivilegeCheck или SePrivilegeCheck при работе в режиме ядра. API-функции, связанные с привилегиями, ничего не знают о существовании прав учетных записей, а вот API-функции, связанные с правами учетных записей, о наличии привилегий осведомлены.

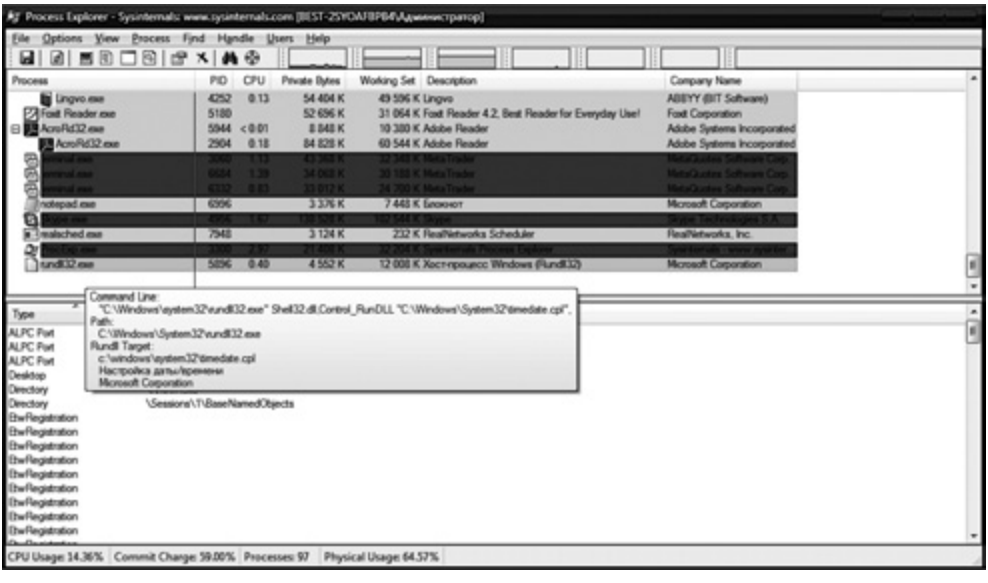
В отличие от прав учетных записей, привилегии могут быть включены или выключены. Чтобы проверка привилегии прошла успешно, она должна присутствовать в указанном маркере и быть включена. В основу этой схемы положена идея о том, что привилегии должны быть включены только при их востребованности, чтобы процесс не мог по недосмотру выполнить привилегированную небезопасную операцию.

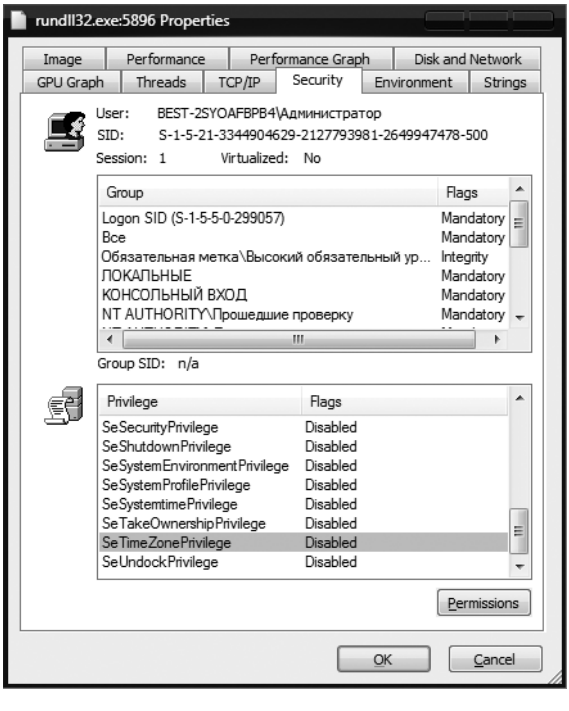
ЭКСПЕРИМЕНТ: НАБЛЮДЕНИЕ ЗА ВКЛЮЧЕНИЕМ ПРИВИЛЕГИИ

Выполняя следующие действия, можно увидеть, что утилита Дата и время (Date and Time Control Panel) включает привилегию SeTimeZonePrivilege в ответ на ваше использование ее интерфейса для изменения часового пояса компьютера:

1. Запустите Process Explorer и установите частоту обновления на паузу (Paused).
2. Щелкните правой кнопкой мыши на часах в области уведомлений и выберите пункт Настройка даты и времени (Adjust Date/Time). При принудительном обновлении с помощью клавиши F5 появится выделенный зеленым цветом новый процесс Rundll32.
3. Поставьте указатель мыши на процессе Rundll32 и проверьте, что цель (target) содержит текст «Настройка даты/времени» («Time Date Control Panel Applet»), а также указан путь к Timedate.cpl. Присутствие этого аргумента заставляет Rundll32, которая является DLL-библиотекой Панели управления, загрузить DLL-библиотеку, реализующую пользовательский интерфейс, позволяющий вам изменить дату и время.

4. Посмотрите на содержимое вкладки Security (Безопасность) диалогового окна Properties (Свойства) вашего процесса Rundll32. Вы должны увидеть, что привилегия SeTimeZonePrivilege выключена.





5. Теперь щелкните на кнопке Изменить часовой пояс (Change Time Zone) в окне утилиты Панели управления, закройте относящееся к процессу

диалоговое окно Properties (Свойства), а затем откройте его снова. Теперь на вкладке Security (Безопасность) вы должны увидеть, что привилегия SeTimeZonePrivilege включена.

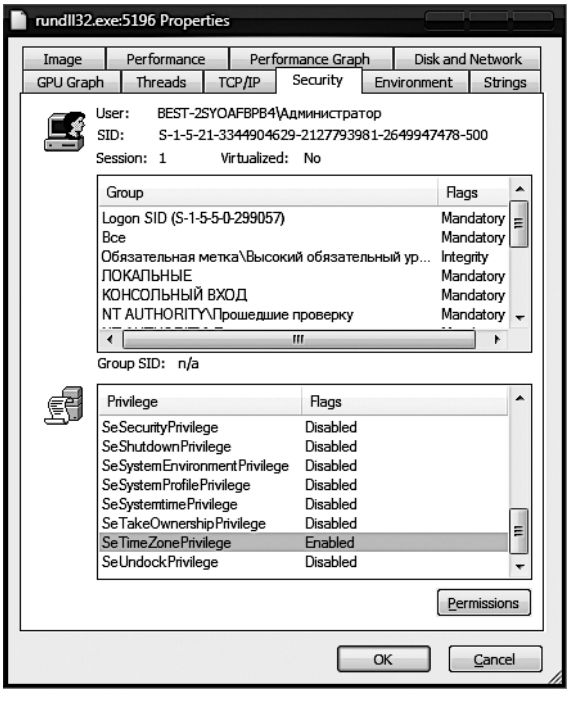


Таблица 6.9. Привилегии

Привилегия	Право пользователя	Использование привилегии
SeAssign-PrimaryToken-Privilege	Заменять маркер уровня маркера	Проверяется различными компонентами, например функцией NtSetInformationJob, устанавливающей маркер процесса
SeAudit-Privilege	Генерировать аудиты безопасности	Нужна для создания событий с помощью API-функции ReportEvent для журнала событий безопасности
SeBackup-Privilege	Создавать резервные копии файлов и каталогов	Заставляет NTFS предоставить следующие виды доступа к любому файлу или каталогу, не обращая внимание на имеющийся дескриптор безопасности: READ_CONTROL, ACCESS_SYSTEM_SECURITY, FILE_GENERIC_READ, FILE_TRAVERSE. Следует заметить, что при открытии файла для резервного копирования вызывающий процесс должен указать флаг FILE_FLAG_BACKUP_SEMANTICS. При использовании функции RegSaveKey допускает также соответствующий доступ к разделам реестра

продолжение ⇩

Таблица 6.9 *(продолжение)*

Привилегия	Право пользователя	Использование привилегии
SeChange-NotifyPrivilege	Обходить промежуточные проверки	Используется NTFS, чтобы избежать проверки разрешений на промежуточные каталоги при многоуровневом поиске каталога. Также используется файловыми системами, когда приложения регистрируются на уведомления об изменениях в структуре файловой системы
SeCreate-Global-Privilege	Создавать глобальные объекты	Требуется процессу для создания объектов разделов и символических ссылок в каталогах пространства имен диспетчера объектов, назначенных для другого сеанса, отличающегося от сеанса вызывающего процесса
SeCreate-Pagefile-Privilege	Создавать файлы подкачки	Проверяется функцией NtCreatePagingFile, используемой для создания новой страницы подкачки
SeCreate-Permanent-Privilege	Создавать постоянные общие объекты	Проверяется диспетчером объектов при создании постоянного объекта (который не будет перемещен при отсутствии на него каких-либо ссылок)
SeCreate-SymbolicLink-Privilege	Создавать символические ссылки	Проверяется NTFS при создании символических ссылок в файловой системе с помощью API-функции CreateSymbolicLink
SeCreate-TokenPrivilege	Создавать объект маркера	Эта привилегия проверяется функцией NtCreateToken, создающей объект маркера
SeDebug-Privilege	Проводить отладку программ	Если эта привилегия у вызывающего процесса включена, диспетчер процессов разрешает доступ к любому процессу или потоку с использованием функции NtOpenProcess или функции NtOpenThread, независимо от дескриптора безопасности процесса или потока (исключение составляют защищенные процессы)
SeEnable-Delegation-Privilege	Облекать доверием для делегирования компьютер и учетную запись пользователя	Используется службами Active Directory для делегирования подтвержденных полномочий
SeImperso-natePrivilege	Заимствовать права клиента после аутентификации	Проверяется диспетчером процессов, когда потоку нужно воспользоваться маркером для заимствования прав, а маркер представляет другого пользователя, чем тот, который указан в маркере процесса данного потока
SeIncrease-BasePriority-Privilege	Поднимать привилегии планирования	Проверяется диспетчером процессов и требует повышения привилегий процесса
SeIncrease-QuotaPrivilege	Регулировать квоты памяти для процесса	Приводится в исполнение при изменении пороговых величин рабочего набора процесса, квот пулов выгружаемой и невыгружаемой памяти процесса и квоты времени использования центрального процессора

Привилегия	Право пользователя	Использование привилегии
SeIncrease-WorkingSet-Privilege	Увеличивать рабочий набор процесса	Требуется для вызова функции SetProcessWorkingSetSize для увеличения минимума рабочего набора. Это опосредованно позволяет процессу блокировать с помощью функции VirtualLock минимальный рабочий набор памяти
SeLoadDriver-Privilege	Загружать и выгружать драйверы устройств	Проверяется функциями драйверов NtLoadDriver и NtUnloadDriver
SeLock-Memory-Privilege	Блокировать страницы в памяти	Проверяется функцией NtLockVirtualMemory, являющейся реализацией функции VirtualLock в режиме ядра
SeMachine-Account-Privilege	Добавлять рабочие станции к домену	Проверяется диспетчером учетных записей безопасности на контроллере домена при создании в домене учетной записи машины
SeManage-Volume-Privilege	Выполнять задачи обслуживания томов	Приводится в исполнение драйверами файловой системы в ходе операций открытия тома, которая требуется для осуществления проверки диска и проведения дефрагментации
SeProfile-SingleProcess-Privilege	Профилировать отдельный процесс	Проверяется супервыборкой — Superfetch и предварительной выборкой при запросе информации для отдельного процесса с помощью API-функции NtQuerySystemInformation
SeRelabel-Privilege	Модифицировать метку объекта	Проверяется SRM при повышении уровня целостности объекта, чьим владельцем является другой пользователь, или при попытке повысить уровень целостности того объекта, чей уровень выше, чем у маркера вызывающего процесса
SeRemote-Shutdown-Privilege	Принудительно завершать работу с удаленной системы	Служба Winlogon проверяет, что удаленные программы, вызывающие функцию InitiateSystemShutdown, обладают данной привилегией
SeRestore-Privilege	Восстанавливать файлы и каталоги	Эта привилегия заставляет NTFS предоставлять следующие виды доступа любому файлу или каталогу, независимо от имеющегося дескриптора безопасности: WRITE_DAC WRITE_OWNER ACCESS_SYSTEM_SECURITY FILE_GENERIC_WRITE FILE_ADD_FILE FILE_ADD_SUBDIRECTORY DELETE Следует иметь в виду, что при открытии файла для восстановления вызывающий процесс должен указать флаг FILE_FLAG_BACKUP_SEMANTICS. При использовании функции RegSaveKey допускает также соответствующий доступ к разделам реестра

продолжение ⇩

Таблица 6.9 *(продолжение)*

Привилегия	Право пользователя	Использование привилегии
SeSecurity-Privilege	Управлять аудитом и журналом безопасности	Требуется для получения доступа к SACL дескриптора безопасности, а также для чтения и очистки журнала событий безопасности
SeShutdown-Privilege	Завершать работу системы	Эта привилегия проверяется функциями NtShutdownSystem и NtRaiseHardError, присутствующими в системном диалоговом окне в интерактивной консоли
SeSyncAgent-Privilege	Синхронизировать данные службы каталогов	Требуется для использования в службах синхронизации каталога LDAP. Позволяет владельцу читать все объекты и свойства в каталоге, независимо от защиты объектов и свойств
SeSystem-Environment-Privilege	Модифицировать среду переменных встроенного программного обеспечения	Требуется функциям NtSetSystemEnvironment Value и NtQuerySystemEnvironment Value для изменения и чтения переменных среды встроенного программного обеспечения с использованием уровня аппаратных абстракций — hardware abstraction layer (HAL)
SeSystem-Profile-Privilege	Профилировать производительность системы	Проверяется функцией NtCreateProfile, используемой для профилирования системы. Используется, к примеру, утилитой Kernprof
SeSystemtime-Privilege	Изменять системное время	Требуется для изменения времени или даты
SeTake-Ownership-Privilege	Становиться владельцем файлов и других объектов	Требуется для приобретения прав владения объектом без предоставления избирательного доступа
SeTcbPrivilege	Действовать в качестве части операционной системы	Проверяется монитором безопасности, когда в маркере установлен ID сеанса диспетчером устройств Plug and Play для создания событий Plug and Play и управления ими, функцией BroadcastSystemMessageEx, при вызове с параметром BSM_ALLDESKTOPS, функцией LsaRegisterLogonProcess и когда при вызове функции NtSetInformationProcess указывается приложение, используемое в качестве VDM
SeTimeZone-Privilege	Изменять часовой пояс	Требуется для изменения часового пояса
SeTrusted-CredMan-Access-Privilege	Получать доступ к диспетчеру учетных данных в качестве доверенного вызывающего	Проверяется диспетчером учетных данных, чтобы убедиться в том, что он должен доверять вызывающему процессу с информацией об учетных данных, которая может быть запрошена в виде обычного текста. Предоставляется по умолчанию только службе Winlogon

Привилегия	Право пользователя	Использование привилегии
SeUndock-Privilege	Отключать компьютер от док-станции	Проверяется диспетчером устройств Plug and Play, работающим в пользовательском режиме, когда либо инициируется отключение компьютера от док-станции, либо делается запрос на извлечение устройства
SeUnsolicited-InputPrivilege	Получать невостребованные данные от терминального устройства	Эта привилегия в настоящее время в Windows не используется

ЭКСПЕРИМЕНТ: ПРИВИЛЕГИЯ ОБХОДА ПРОМЕЖУТОЧНЫХ ПРОВЕРОК

Системные администраторы должны знать о привилегии обхода промежуточных проверок — Bypass Traverse Checking (которая внутри системы называется SeNotifyPrivilege) и последствиях ее использования. В данном эксперименте демонстрируется, как непонимание ее поведения может привести к нарушению режима безопасности.

1. Создайте папку, а внутри этой папки создайте новый текстовый файл с каким-нибудь текстом.
2. Перейдите в Explorer к новому файлу и откройте в его диалоговом окне Свойства (Properties) вкладку Безопасность (Security). Щелкните на кнопке Дополнительно (Advanced) и снимите флажок, управляющий наследованием. Когда появится предложение удалить или скопировать права наследования, выберите Копировать (Copy).
3. Теперь измените безопасность новой папки, чтобы у вашей учетной записи не было никакого доступа к папке. Для этого выберите свою учетную запись, а затем выберите все флажки Запретить (Deny) в списке разрешений.
4. Запустите Блокнот и через меню Файл (File) перейдите в диалоговое окно Открыть (Open). Доступ к новому каталогу должен быть запрещен.
5. В поле Имя файла (File Name) диалогового окна Открыть (Open) наберите полный путь к новому файлу. Файл должен открыться.

Если у вашей учетной записи нет привилегии обхода промежуточных проверок, NTFS выполняет проверку доступа к каждому каталогу пути при попытке открыть файл, что в данном примере приведет к запрещению доступа к файлу. ■

Суперпривилегии

Некоторые привилегии настолько влиятельны, что пользователь, которому они назначены, фактически является «суперпользователем», имеющим полный контроль над компьютером. Эти привилегии могут использоваться несчетным количеством способов для получения несанкционированного доступа к неограниченным в иной ситуации ресурсам и к выполнению несанкционированных операций. Но мы сосредоточимся на использовании привилегии выполнения кода, предоставляющего пользователю не назначенные ему привилегии, имея в виду, что этой возможностью можно воспользоваться для выполнения на локальной машине любой нужной пользователю операции.

В этом разделе дается перечень привилегий и рассматриваются способы их возможных применений. Другие привилегии, например блокировка страниц в физической памяти — Lock Pages In Physical Memory, — могут применяться в системе для атак «отказа в обслуживании» (denial-of-service attacks), но такие привилегии здесь не рассматриваются. Следует заметить, что системы с включенным контролем учетных записей (UAC) такие привилегии будут предоставлять только приложениям, запущенным на высоком уровне целостности (high) или выше, даже если учетная запись ими обладает:

- **Проводить отладку программ** (Debug programs). Пользователь с этой привилегией может открывать любой процесс в системе (за исключением защищенного — Protected Process), не обращая внимания на имеющийся у процесса дескриптор безопасности. Пользователь может выполнять программу, открывающую LSASS-процесс, например копировать исполняемый код в его адресное пространство, а затем, с помощью Windows API-функции CreateRemoteThread, вводить поток для выполнения введенного кода в более привилегированном контексте безопасности. Код может предоставлять пользователю дополнительные привилегии и членство в группах.
- **Приобретать права владения** (Take ownership). Эта привилегия позволяет ее владельцу приобретать права владения любым защищаемым объектом (даже защищенными потоками и процессами) путем записи своего собственного SID в поле владельца (owner) дескриптора безопасности объекта. Следует напомнить, что владельцу всегда предоставляются права на чтение и внесение изменений в DACL дескриптора безопасности, поэтому процесс с этой привилегией может вносить изменения в DACL, предоставляя самому себе полный доступ к объекту, а затем закрывать и снова открывать объект с полным доступом. Это позволит владельцу просматривать конфиденциальные данные и даже заменять системные файлы, выполняющие часть обычной системной операции, например LSASS, своими собственными программами, предоставляющими повышенные пользовательские привилегии.
- **Восстанавливать файлы и каталоги** (Restore files and directories). Пользователь, получивший эту привилегию, может заменять любые файлы в системе своими собственными файлами. Он может воспользоваться этой властью путем замены системных файлов, как описывалось в предыдущем пункте.
- **Загружать и выгружать драйверы устройств** (Load and unload device drivers). Злоумышленник может воспользоваться этой привилегией для загрузки в систему драйверов устройств. Эти драйверы рассматриваются в качестве доверенной части операционной системы, которые могут выполняться внутри этой системы с правами учетной записи System, поэтому драйвер может запускать привилегированные программы, назначающие пользователю дополнительные права.
- **Создавать объект маркера** (Create a token object). Эта привилегия может использоваться в соответствии с предназначением для создания маркеров, представляющих произвольные учетные записи пользователей с произвольной принадлежностью к той или иной группе и назначенными привилегиями.
- **Действовать в качестве части операционной системы** (Act as part of operating system). Эта привилегия проверяется функцией LsaRegisterLogonProcess, которую процесс вызывает для установки доверенного подключения к LSASS.

Злоумышленник, обладающий данной привилегией, может установить доверенное подключение к LSASS, а затем выполнить функцию LsaLogonUser, которая используется для создания новых сеансов входа в систему. Функция LsaLogonUser требует подходящего имени пользователя и пароля, а в качестве дополнительного параметра принимает список SID-идентификаторов, которые добавляются этой функцией к исходному маркеру, создаваемому для нового сеанса входа в систему.

Следовательно, пользователь может воспользоваться своим собственным именем и паролем для создания нового сеанса входа в систему, включающего SID-идентификаторы с большим количеством привилегированных групп или пользователей в получающемся в результате этого маркере.

Следует заметить, что использование повышенных привилегий за границы машины не выходит, и на сеть не распространяется, поскольку любое взаимодействие с другим компьютером требует аутентификации в контроллере домена и проверки доменных паролей, которые не хранятся на компьютере ни в виде простого текста, ни в зашифрованном виде, поэтому вредоносному коду они не доступны.

Маркеры доступа процессов и потоков

Понятия, рассмотренные в данной главе, объединены на рис. 6.9, где показаны основные структуры безопасности процесса и потока. На рисунке показано, что

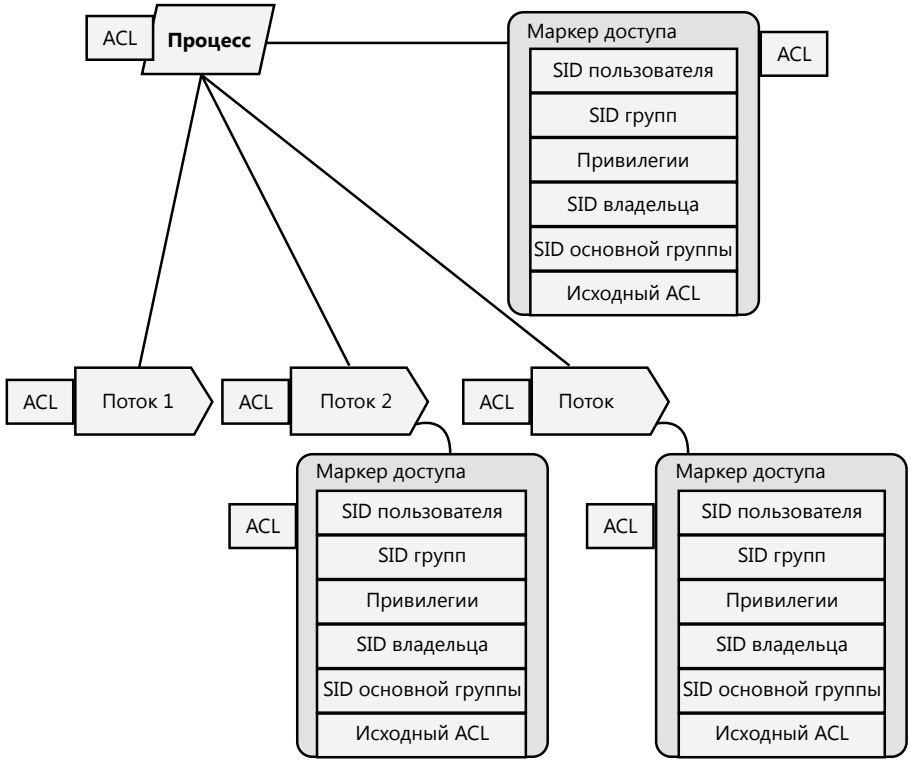


Рис. 6.9. Структуры безопасности процесса и потоков

у объекта процесса и у объектов потоков имеются ACL-списки, которые также имеются и у самих маркеров доступа. Также на этом рисунке поток 2 и поток 3 имеют маркеры заимствования прав, а поток 1 использует исходный маркер доступа процесса.

Аудит безопасности

Диспетчер объектов может генерировать события аудита в качестве результата проверки доступа, а Windows-функции, доступные пользовательским приложениям, могут генерировать эти события напрямую. Код режима ядра всегда позволяет генерировать событие аудита. С аудитом связаны две привилегии: SeSecurityPrivilege и SeAuditPrivilege. Для управления журналом событий безопасности и для просмотра или настройки SACL объекта у процесса должна быть привилегия SeSecurityPrivilege. Но процессы, вызывающие системные службы аудита для успешного генерирования записи аудита, должны обладать привилегией SeAuditPrivilege.

Политика аудита локальной системы управляет решением на аудит конкретного типа события безопасности. Политика аудита, также называемая локальной политикой безопасности, является одной частью политики безопасности LSASS, обеспечиваемой на локальной системе, и настраивается с помощью редактора локальной политики безопасности, показанного на рис. 6.10.

Конфигурация политики аудита (как основные настройки в разделе локальных политик, так и расширенная конфигурация политики аудита, рассматриваемая далее) хранится в реестре в качестве битового значения в разделе HKEY_LOCAL_MACHINE\SECURITY\Policy\PolAdtEv.

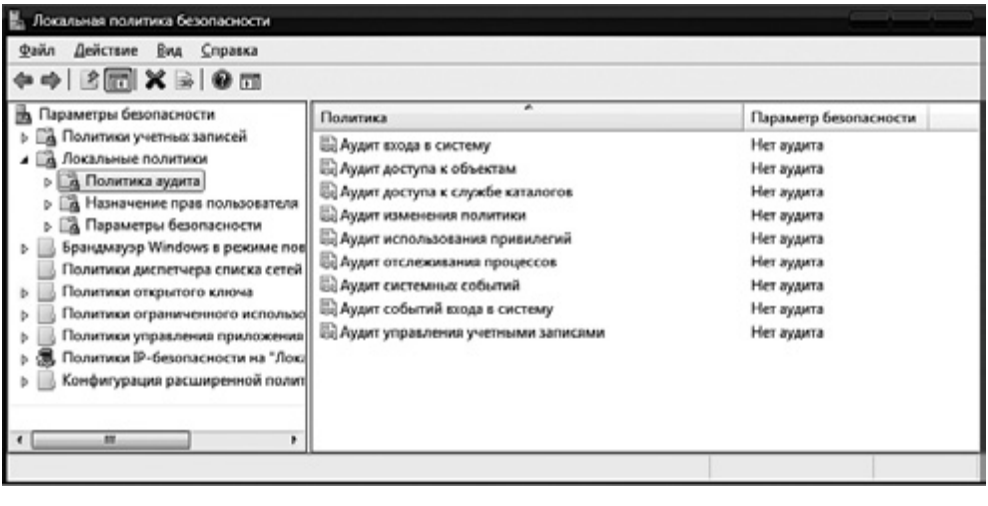


Рис. 6.10. Конфигурация редактора политики локальной безопасности

LSASS отправляет сообщения SRM, чтобы проинформировать его о политике аудита в ходе инициализации системы, а потом при изменениях политики. LSASS отвечает за получение записей аудита, сгенерированных на основе событий аудита от SRM, редактирует записи и отправляет их регистратору событий

(Event Logger). LSASS (вместо SRM) отправляет эти записи потому, что добавляет к ним вполне уместные подробности, например информацию, необходимую для более полной идентификации подвергаемого аудиту процесса.

SRM отправляет записи аудита через свое подключение ALPC к LSASS. Затем регистратор событий помещает запись аудита в журнал регистрации событий безопасности. Кроме записей аудита, передаваемых SRM, и LSASS, и SAM генерируют записи аудита, которые LSASS отправляет непосредственно регистратору событий, а API-функции AuthZ позволяют приложениям генерировать аудиты, определяемые самими приложениями. Эти общие потоки показаны на рис. 6.11.

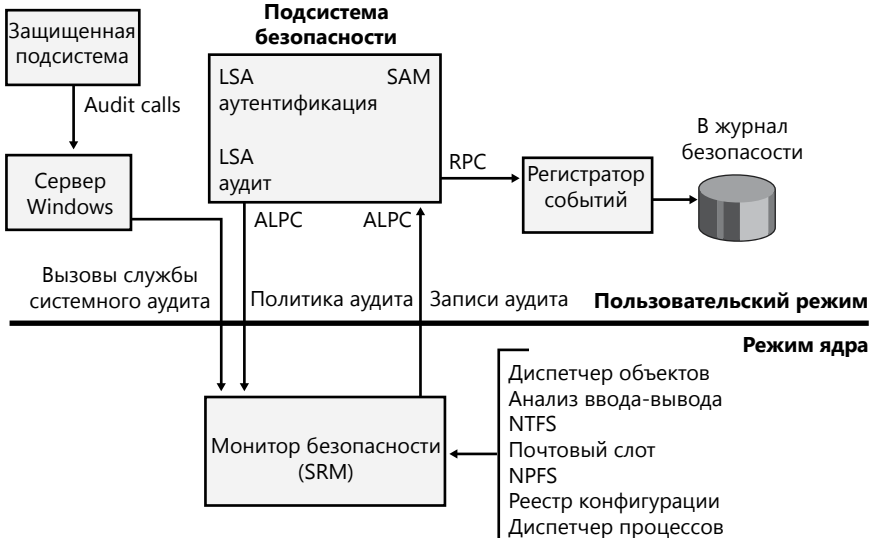


Рис. 6.11. Поток записей аудита безопасности

По мере получения записи аудита ставятся в очередь на отправку LSA, они не передаются пакетами. Записи аудита передаются от SRM к подсистеме безопасности одним из двух возможных способов. Если запись аудита невелика по размеру (меньше чем максимальный размер сообщения ALPC), она отправляется как ALPC-сообщение. Записи аудита копируются из адресного пространства SRM в адресное пространство процесса LSASS. Если запись аудита имеет большой размер, SRM использует общую память, чтобы сообщение было доступно LSASS, и просто передает указатель в ALPC-сообщении.

Аудит доступа к объекту

Важной областью применения механизма аудита во многих средах окружения является ведение журнала доступов к защищенным объектам, в частности к файлам. Для этого должна быть включена политика Аудит доступа к объектам (Audit Object Access) и в системных списках управления доступом должны быть ACE-элементы аудита, разрешающие проведение аудита в интересующих объектах.

Когда средство доступа пробует открыть дескриптор объекта, монитор безопасности сначала определяет, разрешена или запрещена подобная попытка. Если

включен аудит доступа к объекту, SRM затем сканирует системный ACL объекта. Есть два типа ACE-элементов аудита: доступ разрешен и доступ запрещен. ACE аудита должен соответствовать любому идентификатору безопасности, имеющемуся у средства доступа, он должен соответствовать любому из запрошенных методов доступа, и его тип (доступ разрешен или доступ запрещен) должен соответствовать результату проверки доступа, чтобы была сгенерирована запись аудита доступа к объекту.

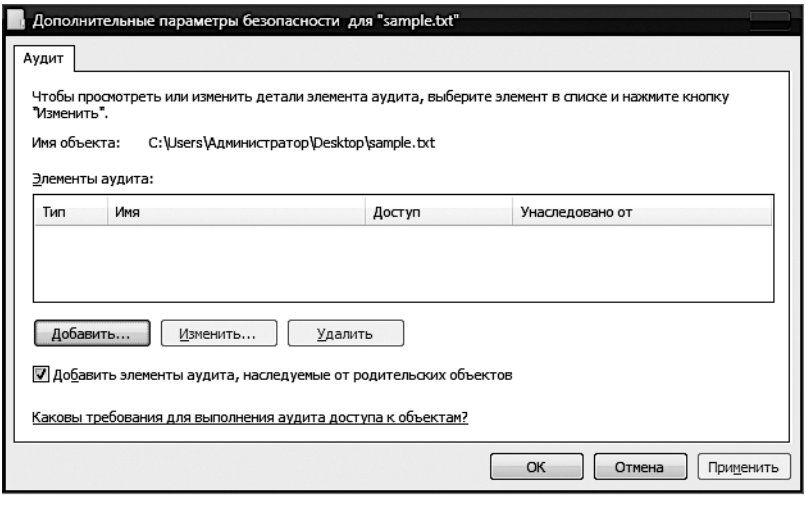
Записи аудита доступа к объекту включают не только сам факт разрешенного или запрещенного доступа, но также и причину успеха или отказа. Эта «причина для доступа», дающая отчет в общем виде, принимает в записи аудита форму записи управления доступом, указанной в языке определения дескриптора безопасности — SDDL (Security Descriptor Definition Language). Это позволяет проводить диагностику сценариев, в которых объекту, к которому по вашему убеждению доступ был запрещен, этот доступ был разрешен, или наоборот, путем идентификации определенных записей управления доступом, ставших причиной того, что попытка доступа удалась или провалилась.

На рис. 6.10 можно заметить, что проведение аудита доступа к объекту изначально выключено (что касается и всех остальных политик аудита).

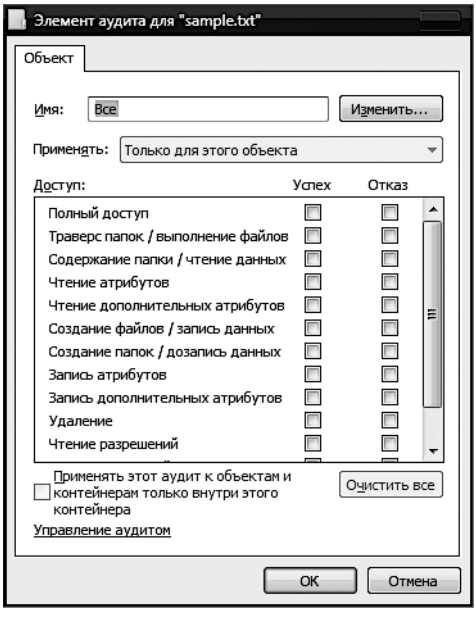
ЭКСПЕРИМЕНТ: ПРОВЕДЕНИЕ АУДИТА ДОСТУПА К ОБЪЕКТУ

Продемонстрировать аудит доступа к объекту можно с помощью следующих действий:

1. Перейдите в Explorer к файлу, к которому в обычных условиях доступ разрешен. В диалоговом окне Свойства (Properties) этого файла щелкните на вкладке Безопасность (Security), а затем щелкните на кнопке Дополнительно (Advanced). Щелкните на вкладке Аудит (Auditing) и пройдите через предупреждение о необходимости административных привилегий. Появившееся в результате этого диалоговое окно позволяет вам добавлять в системный список управления доступом — System Access Control List — этого файла записи об аудите управления доступом.



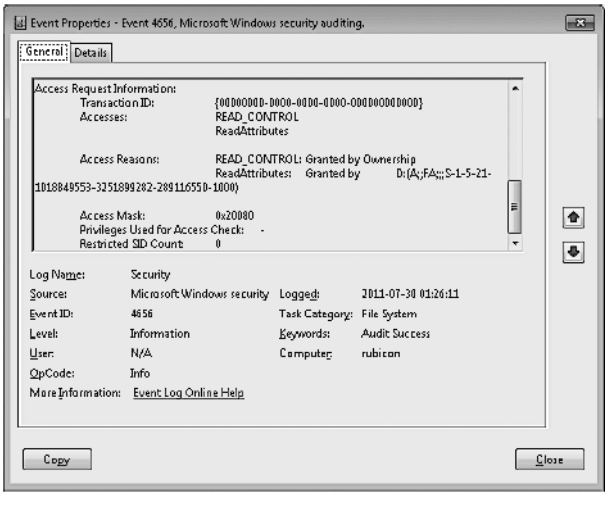
2. Щелкните на кнопке Добавить (Add). В появившемся диалоговом окне Выбор «Пользователь» или «Группа» (Select User Or Group) введите свое собственное имя пользователя и пароль или название группы, в которой состоите, например Все (Everyone), и щелкните на кнопке Проверить имена (Check Names), а затем на кнопке OK. В результате появится диалоговое окно для создания элемента аудита (Auditing Access Control Entry) этого файла при доступе к нему со стороны пользователя или группы.



3. В столбце Успех (Successful) выберите Полный доступ (Full control) (что приведет также к выбору всех остальных методов доступа). Щелкните четыре раза на кнопке OK, чтобы закрыть диалоговое окно Свойства (Properties).
4. Находясь в Explorer, дважды щелкните на значке файла, чтобы открыть его в связанной с ним программе.
5. В утилите просмотра событий — Event Viewer — перейдите к журналу безопасности — Security log. Обратите внимание, что там нет записи для доступа к файлу. Причина в том, что политика аудита для доступа к объекту еще не настроена.
6. В редакторе локальной политики безопасности перейдите к разделу Локальные политики (Local Policies), Политика аудита (Audit Policy). Дважды щелкните на записи Аудит доступа к объектам (Audit Object Access), а затем установите флажок Успех (Success), чтобы включить аудит успешного доступа к файлу.
7. В утилите Event Viewer щелкните на пунктах Action (Действие), Refresh (Обновить). Обратите внимание на то, как изменения в политике аудита повлияли на записи аудита.
8. Перейдя в Explorer, дважды щелкните на значке файла, чтобы открыть его еще раз.

9. В утилите Event Viewer щелкните на пунктах Action (Действие), Refresh (Обновить). Обратите внимание на то, что теперь появились несколько записей аудита доступа к файлу.

Найдите одну из записей аудита доступа к файлу, у которой идентификатор события Event ID имеет значение 4656, он проявляется в качестве «дескриптора объекта, который был запрошен». Прокрутите вниз содержимое текстового поля и найдите раздел Access Reasons. В следующем примере показано, что были запрошены два метода доступа: READ_CONTROL и ReadAttributes. Первый из них был предоставлен потому, что запрашивающий доступ был владельцем файла, а последний был предоставлен по причине показанного элемента управления доступом — Access Control Entry. Этот ACE включает SID пользователя, предпринявшего попытку доступа, и включает назначение A:FA, обозначающее, что этому SID разрешены — Allowed (A) — все методы доступа к файлу — all file access methods (FA).



Глобальная политика аудита

В дополнение к ACE-элементам доступа к объектам, применяемым в отношении отдельных объектов, в отношении системы может быть определена глобальная политика аудита, позволяющая проводить аудит доступа к объекту для всех объектов файловой системы, для всех разделов реестра или и к тем и к другим. Поэтому аудитор безопасности может быть уверен, что нужный аудит будет выполнен без необходимости установки или изучения списков SACL у всех отдельных интересующих объектов.

Администратор может установить или запросить глобальную политику аудита с помощью команды AuditPol с ключом /resourceSACL. То же самое можно сделать с помощью программы, вызывающей API-функции AuditSetGlobalSacl и AuditQueryGlobalSacl. Как и в случае изменений SACL-списков объектов, изменение этих глобальных SACL-списков требует привилегии SeSecurityPrivilege.

ЭКСПЕРИМЕНТ: УСТАНОВКА ПОЛИТИКИ ГЛОБАЛЬНОГО АУДИТА

Для включения политики глобального аудита можно воспользоваться командой AuditPol.

1. Если это еще не сделано в предыдущем эксперименте, в редакторе локальной политики безопасности перейдите к настройкам политики аудита (как показано на рис. 6.10), дважды щелкните на пункте Аудит доступа к объектам (Audit Object Access) и включите аудит, как для успеха, так и для отказа. Учтите, что на большинстве систем SACL-списки, определяющие аудит доступа к объектам, встречаются довольно редко, поэтому на данный момент времени записей будет не так уж и много, если там вообще будут какие-нибудь записи.
2. В окне командной строки с повышенными привилегиями введите следующую команду:

```
C:\> auditpol /resourceSACL
```

   Она выдаст краткую информацию о командах для настройки и запроса политики глобального аудита.

3. В том же самом окне командной строки с повышенными привилегиями введите следующие команды:

```
C:\> auditpol /resourceSACL /type:File /view
C:\> auditpol /resourceSACL /type:Key /view
```

   На обычной системе каждая из этих команд сообщит об отсутствии глобального списка Global SACL для соответствующего типа ресурсов. (Учтите, что ключевые слова «File» и «Key» чувствительны к регистру букв.)

4. В том же самом окне командной строки с повышенными привилегиями введите следующую команду:

```
C:\> auditpol /resourceSACL /set /type:File /user:yourusername /success /failure
/access:FW
```

   В результате ее выполнения глобальная политика аудита будет настроена таким образом, что все попытки открытия указанным пользователем файлов с доступом по записи (FW) будут приводить к появлению записей аудита как при удачной, так и при неудавшейся попытке открытия файла. Именем пользователя должно быть конкретное имя пользователя в системе, группа (например, Все (Everyone)), определенное доменом имя пользователя, вида *имя_домена**имя_пользователя* или SID.

5. Работая под указанным именем пользователя, воспользуйтесь Explorer или другой утилитой, для того чтобы открыть файл. Затем загляните в журнал безопасности в системном журнале событий и найдите в нем записи аудита.
6. В завершение эксперимента воспользуйтесь командой auditpol для удаления глобального списка SACL, созданного при выполнении пункта 4:

```
C:\> auditpol /resourceSACL /remove /type:File /user:yourusername
```

■

Глобальная политика аудита хранится в реестре в виде пары системных списков управления доступом в HKEY_LOCAL_MACHINE\SECURITY\Policy\

GlobalSaclNameFile и в HKEY_LOCAL_MACHINE\SECURITY\Policy\GlobalSaclNameKey. Эти разделы могут быть изучены путем запуска программы Regedit.exe под учетной записью System (система), в последовательности, изложенной ранее в разделе «Системные компоненты безопасности». Эти разделы не будут присутствовать в реестре, пока хотя бы однажды не будут установлены соответствующие глобальные SACL-списки.

Глобальная политика аудита не может быть отменена путем применения SACL-списков объектов, а вот SACL-списки объектов могут позволить проводить дополнительный аудит. Например, глобальная политика аудита может потребовать проведение аудита доступа по чтению всех пользователей ко всем файлам, а SACL-списки отдельных файлов могут добавить аудит доступа по записи к таким файлам со стороны определенных пользователей или более определенных групп пользователей.

Глобальная политика аудита может также быть сконфигурирована через редактор локальной политики безопасности через Конфигурацию расширенной политики аудита, рассматриваемую в следующем подразделе.

Конфигурация расширенной политики аудита

В дополнение к ранее рассмотренным настройкам политики аудита, редактор локальной политики безопасности предлагает еще более тонко настраиваемый набор средств управления аудитом, показанный на рис. 6.12 под заголовком Конфигурация расширенной политики аудита (Advanced Audit Policy Configuration).

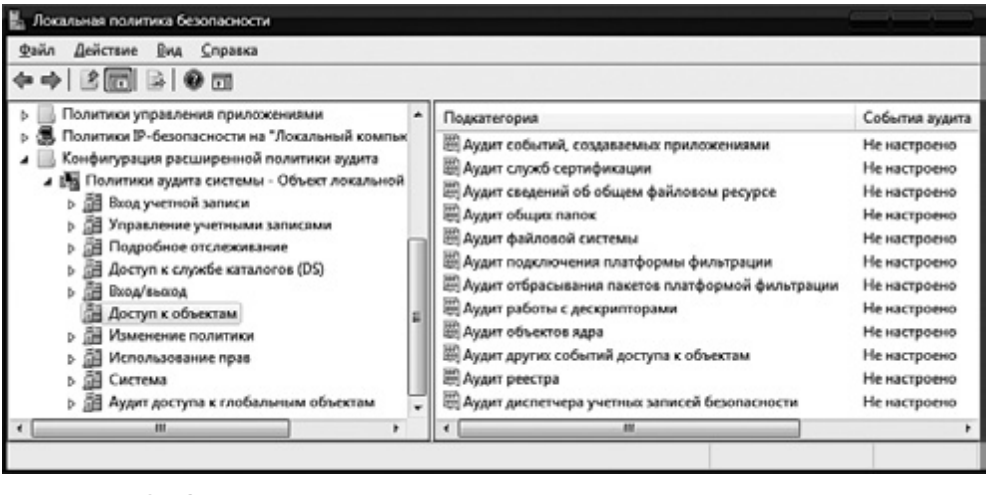


Рис. 6.12. Конфигурация расширенной политики аудита в редакторе локальной политики безопасности

Каждая из девяти настроек политики аудита в разделе Локальные политики, показанных на рис. 6.10, отображается здесь на группу настроек, предоставляющую более тонкое управление. Например, Аудит доступа к объектам в разделе Локальные политики разрешает доступ ко всем подвергаемым аудиту объектам, а здесь можно настроить аудит доступа к объектам различного типа так, чтобы он

управлялся индивидуально. Включение одной из настроек политики аудита в разделе Локальные политики косвенным образом включает все соответствующие расширенные события политики аудита, но если требуется более тонкое управление над контентами журнала аудита, расширенные настройки могут быть установлены индивидуальным образом. Затем стандартные настройки становятся продуктом расширенных настроек, но это нельзя увидеть в редакторе локальной политики безопасности. Попытки указать настройки аудита с использованием как основных, так и расширенных настроек могут привести к неожиданным результатам.

Подраздел Аудит доступа к глобальным объектам (Global Object Access Auditing) в разделе Конфигурация расширенной политики аудита (Advanced Audit Policy Configuration) может использоваться для настройки глобальных системных списков управления доступом (Global SACL), с использованием графического интерфейса, идентичного тому, который можно увидеть в Explorer или в редакторе реестра для дескрипторов безопасности в файловой системе или в реестре.

Вход в систему

Интерактивный вход в систему (в отличие от сетевого входа) осуществляется посредством взаимодействия с процессом входа в систему (Winlogon), процессом интерфейса входа пользователя в систему (LogonUI) и их поставщиками учетных данных, LSASS, одним или несколькими пакетами аутентификации и SAM или Active Directory. Пакеты аутентификации являются DLL-библиотеками, выполняющими проверки аутентификации. Kerberos является пакетом аутентификации Windows для интерактивного входа в домен, а MSV1_0 является пакетом аутентификации Windows для интерактивного входа на локальный компьютер, для доменных входов на доверенные домены под управлением тех версий Windows, которые предшествовали версии Windows 2000, и для тех случаев, когда недоступен контроллер доменов.

Winlogon является доверенным процессом, отвечающим за управление взаимодействий с пользователем, связанных с безопасностью. Им координируются вход в систему, запуск первого процесса пользователя при входе в систему, обработка выхода из системы и управление рядом других операций, относящихся к безопасности, включая запуск LogonUI для ввода паролей при входе в систему, изменении паролей и блокировке и разблокировке рабочей станции. Процесс Winlogon должен гарантировать, что операции, связанные с безопасностью, невидимы любым другим активным процессам. Например, Winlogon гарантирует, что не пользующийся доверием процесс не может получить управление рабочим столом в ходе одной из таких операций, получив тем самым доступ к паролю.

Winlogon при получении имени и пароля учетной записи пользователя зависит от установленных в системе поставщиков учетных записей. Эти поставщики являются COM-объектами, которые находятся внутри DLL-библиотек. Исходными поставщиками являются %SystemRoot%\System32\authui.dll и %SystemRoot%\System32\SmartcardCredentialProvider.dll, и они поддерживают как пароль, так и PIN аутентификации смарткарты. Разрешение установки других поставщиков учетных данных позволяет Windows использовать различные механизмы идентификации пользователей. Например, сторонний разработчик может предоставить поставщика учетных записей, который использует

для идентификации пользователей устройство распознавания отпечатка пальца и извлекает пароли этих пользователей из зашифрованной базы данных.

Чтобы защитить адресное пространство Winlogon от ошибок поставщиков учетных данных, которые могут привести к сбою процесса Winlogon (что, в свою очередь, приведет к системному сбою, поскольку Winlogon считается критическим системным процессом), для фактической загрузки поставщиков учетных данных и демонстрации пользователю Windows-интерфейса входа в систему используется отдельный процесс LogonUI.exe. Этот процесс запускается по запросу, как только Winlogon требуется присутствие пользовательского интерфейса, а выход из него осуществляется после завершения нужного действия. Это также позволяет Winlogon просто перезапустить новый процесс LogonUI в случае сбоя, возникшего по какой-либо причине.

Winlogon является единственным процессом, перехватывающим запросы на вход в систему с клавиатуры, которые отправляются через RPC-сообщения из Win32k.sys. Winlogon тут же запускает приложение LogonUI, чтобы вывести пользователю интерфейс входа в систему. После получения от поставщика учетных данных имени пользователя и пароля Winlogon вызывает LSASS, чтобы аутентифицировать попытку пользователя войти в систему. Если пользователь аутентифицирован, процесс входа в систему активирует оболочку входа в систему от имени этого пользователя. Взаимодействие между компонентами, участвующими во входе в систему, показано на рис. 6.13.

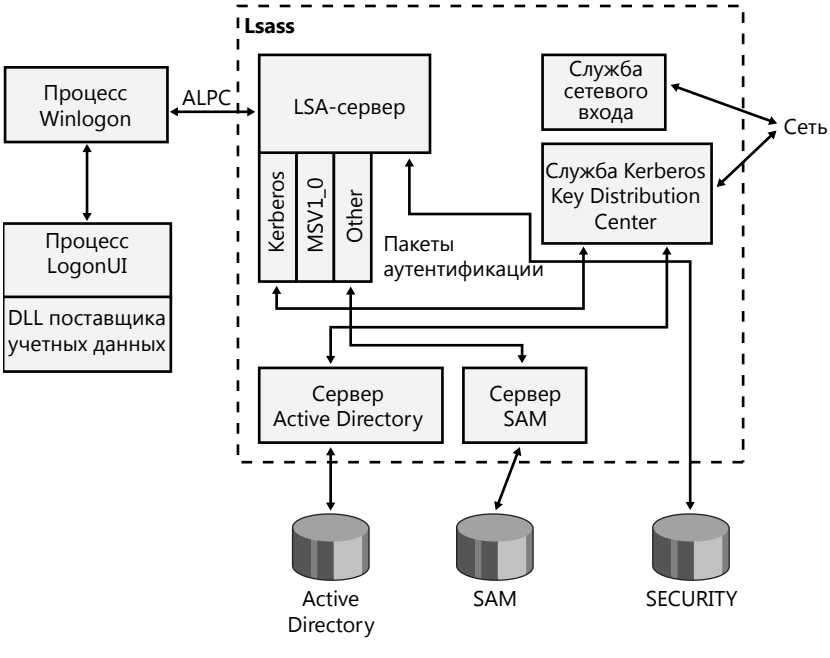


Рис. 6.13. Компоненты, участвующие во входе в систему

Кроме поддержки альтернативных поставщиков учетных данных, LogonUI может загрузить дополнительные DLL-библиотеки сетевого поставщика, необходимые для проведения вторичной аутентификации. Эта возможность позволяет

нескольким сетевым поставщикам собрать всю информацию, касающуюся идентификации и аутентификации в ходе обычного входа в систему. Пользователь, входящий в систему Windows, может одновременно быть аутентифицирован на сервере UNIX. Затем этот пользователь сможет получить доступ к ресурсам сервера UNIX с Windows-машины без необходимости прохождения дополнительной аутентификации. Такая возможность известна как одна из форм единого входа.

Инициализация Winlogon

В ходе инициализации системы, перед тем как станет активным любое пользовательское приложение, Winlogon выполняет следующие действия, чтобы убедиться в том, что он управляет рабочей станцией, в то время как система готова к взаимодействию с пользователем:

1. Создает и открывает интерактивную станцию окна (например, \Sessions\1\Windows\WindowStations\WinSta0 в пространстве имен диспетчера объектов) для представления клавиатуры, мыши и монитора. Winlogon создает дескриптор безопасности для станции, у которого имеется только один ACE-элемент, содержащий только один системный SID. Этот уникальный дескриптор безопасности гарантирует, что никакой другой процесс не сможет получить доступ к рабочей станции, пока это не будет явным образом разрешено процессом Winlogon.
2. Создает и открывает два рабочих стола: рабочий стол приложения (\Sessions\1\Windows\WinSta0\Default, известный также как интерактивный рабочий стол) и рабочий стол Winlogon (\Sessions\1\Windows\WinSta0\Winlogon, известный также как защищенный рабочий стол). Защита рабочего стола Winlogon устроена таким образом, что доступ к этому рабочему столу может получить только процесс Winlogon. Другой рабочий стол позволяет иметь доступ к нему как процессу Winlogon, так и пользователям. Такой порядок означает, что в любое время, когда активен рабочий стол Winlogon, никакой другой процесс не имеет доступа ни к какому активному коду или данным, связанным с рабочим столом. Windows использует это свойство для защиты операций безопасности, использующих пароли и блокирование и разблокирование рабочего стола.
3. Перед тем как кто-нибудь зарегистрируется на компьютере, видимым будет рабочий стол Winlogon. После того как пользователь вошел в систему, нажатие комбинации клавиш Ctrl+Alt+Delete переключает рабочий стол с исходного (Default) на Winlogon и запускает LogonUI. (Этим объясняется вопрос, почему все окна на вашем интерактивном рабочем столе кажутся исчезнувшими, когда вы нажимаете комбинацию Ctrl+Alt+Delete, а затем возвращаются назад, когда вы освобождаете диалоговое окно безопасности Windows.) Таким образом, SAS всегда выдает рабочий стол безопасности, управляемый процессом Winlogon.
4. Устанавливает ALPC-подключение к LSASS-порту LsaAuthenticationPort. Это подключение будет использоваться для обмена информацией в процессе входа в систему, выхода из системы и операций с паролями, который будет осуществляться с помощью вызова функции LsaRegisterLogonProcess.
5. Регистрирует Winlogon сервер RPC-сообщений, который прислушивается к уведомлениям SAS, выхода из системы и блокировки рабочей станции

от Win32k. Эта мера не дает троянским программам получать контроль над экраном при вводе комбинации SAS.

ПРИМЕЧАНИЕ

Процесс Wininit (см. главу 3) выполняет действия, подобные рассмотренным в пунктах 1 и 2, позволяя устаревшим интерактивным службам выполняться в сеансе 0, чтобы отображать окна, но он не выполняет никакие другие действия, поскольку сеанс 0 не доступен для входа пользователя в систему.

КАК РЕАЛИЗОВАНА SAS

Безопасность SAS обеспечивается тем, что ни одно приложение не может перехватить комбинацию клавиш Ctrl+Alt+Delete или помешать Winlogon получить эту комбинацию. Win32k.sys резервирует комбинацию клавиш Ctrl+Alt+Delete таким образом, чтобы как только система ввода Windows (реализованная как простой поток ввода в Win32k) видит эту комбинацию, она отправляет RPC-сообщение серверу сообщений Winlogon, который прислушивается к подобным уведомлениям. Комбинация клавиш, отображаемая на зарегистрированные клавиши быстрого вызова, не отправляется никакому другому процессу, кроме того, который ее зарегистрировал, и только поток, зарегистрировавший клавишу быстрого доступа, может отменить регистрацию, поэтому троянские приложения не могут снять регистрацию принадлежности SAS процессу Winlogon.

Windows-функция SetWindowsHook позволяет приложению установить процедуру перехвата, которая вызывается при каждом нажатии комбинации клавиш, даже перед обработкой клавиш быстрого вызова, и она позволяет перехватчику подавлять действие комбинации клавиш. Но код Windows, используемый для обработки клавиш быстрого вызова, содержит особый вариант обработки для Ctrl+Alt+Delete, который отключает перехват, поэтому данная комбинация клавиш не перехватывается. Кроме того, если интерактивный рабочий стол заблокирован, обрабатываются только те клавиши быстрого вызова, которыми владеет процесс Winlogon.

Поскольку рабочий стол Winlogon создается в ходе инициализации, он становится активным рабочим столом. Когда рабочий стол Winlogon активен, он всегда заблокирован. Winlogon разблокирует свой рабочий стол только для переключения на рабочий стол приложения или на рабочий стол заставки. (Заблокировать рабочий стол или снять его блокировку может только процесс Winlogon.)

Этапы входа пользователя в систему

Вход в систему начинается, когда пользователь нажимает комбинацию SAS (Ctrl+Alt+Delete). После нажатия SAS процесс Winlogon запускает процесс LogonUI, который вызывает поставщика учетных данных для получения имени пользователя и пароля. Winlogon также создает для этого пользователя уникальный локальный SID входа в систему, который назначается данному экземпляру рабочего стола (куда входят клавиатура, экран и мышь). Winlogon передает этот SID процессу LSASS в качестве части вызова функции LsaLogonUser. Если вход пользователя в систему пройдет успешно, этот SID будет включен в маркер процесса входа в систему, на этом этапе создается защита доступа к рабочему столу.

Например, еще один вход под той же учетной записью, но в другую систему не сможет вести запись на рабочий стол первой машины, поскольку эта вторая регистрация не попадет в маркер рабочего стола первой регистрации.

После ввода имени пользователя и пароля Winlogon извлекает дескриптор пакета путем вызова LSASS-функции LsaLookupAuthenticationPackage. Пакеты аутентификации перечислены в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\Lsa. Winlogon передает информацию входа в систему пакету аутентификации через функцию LsaLogonUser. Как только пакет проведет аутентификацию пользователя, Winlogon продолжает процесс входа в систему для этого пользователя. Если ни один из пакетов аутентификации не покажет успешного входа в систему, процесс входа прекращается.

Для интерактивного входа в систему Windows использует два стандартных пакета аутентификации: Kerberos и MSV1_0. Исходным пакетом аутентификации на автономной Windows-системе является MSV1_0 (%SystemRoot%\System32\Msv1_0.dll), который реализует протокол LAN Manager 2. LSASS также использует MSV1_0 на компьютерах, являющихся частью домена для аутентификации на доменах и компьютерах с версиями Windows, предшествующими версии Windows 2000, которые не могут определить с целью аутентификации местонахождение доменного контроллера. (Компьютеры, отключенные от сети, относятся к этой последней категории.) Пакет аутентификации Kerberos, который находится в библиотеке %SystemRoot%\System32\Kerberos.dll, используется на компьютерах, входящих в домены Windows. Пакет Windows Kerberos во взаимодействии со службами Kerberos, выполняемыми на контроллере домена, поддерживает протокол Kerberos. Этот протокол основан на Internet RFC 1510. (Подробную информацию о стандарте Kerberos можно найти на веб-сайте Internet Engineering Task Force [IETF] по адресу www.ietf.org.)

Пакет аутентификации MSV1_0 принимает имя пользователя и хэшированную версию пароля и отправляет запрос локальному SAM-администратору для извлечения информации об учетной записи, включающей хэшированный пароль, группы, в которые входит пользователь, и любые ограничения учетной записи. MSV1_0 сначала проверяет ограничения учетной записи, например период времени или тип разрешенного доступа. Если пользователь не может войти в систему по причине ограничений в базе данных SAM, вызов входа в систему завершается отказом, и MSV1_0 возвращает LSA статус ошибки.

Затем MSV1_0 сравнивает хэшированный пароль и имя пользователя с той информацией, которая была получена из SAM. В случае входа в кэшированный домен, MSV1_0 обращается к кэшированной информации с помощью LSASS-функций, которые сохраняют и извлекают «секреты» из базы данных LSA (куст реестра SECURITY). Если информация совпадает, MSV1_0 генерирует LUID для сеанса входа в систему и создает сеанс входа в систему путем вызова LSASS, связывая этот уникальный идентификатор с сеансом и передавая информацию, необходимую для создания в итоге маркера доступа для пользователя. (Следует напомнить, что маркер доступа включает SID пользователя, SID-идентификаторы групп и назначенные привилегии.)

Если пакету MSV1_0 нужна аутентификация с использованием удаленной системы, как при входе пользователя в заслуживающий доверия домен под управлением версии Windows, предшествующей версии Windows 2000, MSV1_0

использует службу Netlogon для обмена данными с экземпляром Netlogon на удаленной системе. Netlogon на удаленной системе взаимодействует с пакетом аутентификации MSV1_0 на этой системе, возвращая результаты аутентификации той системе, на которой выполнялся вход.

ПРИМЕЧАНИЕ

Учтите, что MSV1_0 не кэширует целый хэш пароля пользователя в реестре, поскольку это позволит кому-нибудь, имеющему физический доступ к системе, без особого труда несанкционированно вскрыть доменную учетную запись пользователя и получить доступ к зашифрованным файлам и к сетевым ресурсам, к которым имеет доступ авторизованный пользователь. Вместо этого кэшируется половина хэша. Этой кэшированной половины хэша вполне достаточно для проведения проверки правильности пользовательского пароля, но недостаточно для получения доступа к EFS-ключам и для аутентификации в качестве пользователя домена, поскольку эти действия требуют полного хэша.

Основной управляющий поток для аутентификации Kerberos аналогичен потоку для MSV1_0. Но в большинстве случаев входы в домены выполняются с входящих с них рабочих станций или серверов (а не с контроллера домена), поэтому пакет аутентификации должен вести обмен данными по сети в качестве части процесса аутентификации. Пакет справляется с этим путем обмена данными через порт Kerberos TCP/IP (порт 88) при наличии на контроллере домена службы Kerberos. Служба Kerberos Key Distribution Center (%SystemRoot%\System32\Kdcsvc.dll), которая реализует протокол аутентификации Kerberos, выполняется в LSASS-процессе на контроллерах домена.

После проверки приемлемости хэшированной информации об имени пользователя и пароле с помощью принадлежащих Active Directory объектов учетной записи пользователя, используя %SystemRoot%\System32\Ntdsa.dll, Kdcsvc возвращает LSASS учетные данные домена, в которых по сети той системе, с которой осуществляется вход, возвращаются результаты аутентификации и учетные данные входа пользователя в домен (если вход был успешным).

ПРИМЕЧАНИЕ

Описание аутентификации с использованием Kerberos сильно упрощено, но оно высвечивает роли различных, привлекаемых для этого компонентов. Хотя аутентификация с использованием Kerberos играет ключевую роль в распределенной доменной безопасности в Windows, подробности этого процесса выходят за рамки данной книги.

После того как вход в систему был аутентифицирован, LSASS обращается к базе данных локальной политики за разрешенными пользователю доступами, включая интерактивные, сетевые, пакетные или служебные процессы. Если требуемый вход в систему не соответствует разрешенным доступам, попытка входа в систему будет прекращена. LSASS удаляет только что созданный сеанс входа в систему, очищая любые его структуры данных, а затем возвращая отказ процессу Winlogon, который, в свою очередь, выводит пользователю соответствующее сообщение. Если запрошенный доступ разрешен, LSASS добавляет соответствующие дополнительные идентификаторы безопасности (например, Everyone,

Interactive и т. п.). Затем LSASS проверяет свою базу данных политики на предмет наличия любых предоставленных привилегий для всех SID-идентификаторов для этого пользователя и добавляет эти привилегии к принадлежащему этому пользователю маркеру доступа.

Когда подсистема LSASS аккумулирует всю нужную информацию, она вызывает исполняющую систему для создания маркера доступа. Исполняющая система создает первичный маркер доступа для интерактивного или служебного входа в систему и маркер заимствования для сетевого входа в систему. После успешного создания маркера доступа LSASS дублирует маркер, создавая дескриптор, который может быть передан Winlogon, и закрывает свой собственный дескриптор. Если нужно, операция входа в систему подвергается аудиту. В этот момент LSASS возвращает сообщение об успехе процессу Winlogon наряду с дескриптором маркера доступа, LUID для сеанса входа в систему и профильной информацией, если таковая имеется, которая была возвращена пакетом аутентификации.

ЭКСПЕРИМЕНТ: ВЫВОД СПИСКА АКТИВНЫХ СЕАНСОВ ВХОДА В СИСТЕМУ

Если существует хотя бы один маркер для заданного LUID сеанса входа в систему, Windows рассматривает сеанс входа в систему в качестве активного. Для вывода списка активных сеансов входа в систему можно воспользоваться инструментальным средством LogonSessions из набора Sysinternals, которое использует функцию LsaEnumerateLogonSessions (документированную в Windows SDK):

```
C:\>logonsessions
Logonsesions v1.21
Copyright (C) 2004-2010 Bryce Cogswell and Mark Russinovich
Sysinternals - wwww.sysinternals.com
[0] Logon session 00000000:000003e7:
    User name:    KERNELS\LAPT8$
    Auth package: NTLM
    Logon type:   (none)
    Session:      0
    Sid:          S-1-5-18
    Logon time:   2012-01-16 22:03:38
    Logon server:
    DNS Domain:
    UPN:
[1] Logon session 00000000:0000cf19:
    User name:
    Auth package: NTLM
    Logon type:   (none)
    Session:      0
    Sid:          (none)
    Logon time:   2012-01-16 22:03:38
    Logon server:
    DNS Domain:
    UPN:
[2] Logon session 00000000:000003e4:
```

продолжение ⇩

```
    User name:    KERNELS\LAPT8$
    Auth package: Negotiate
    Logon type:   Service
    Session:      0
    Sid:          S-1-5-20
    Logon time:   2012-01-16 22:03:40
    Logon server:
    DNS Domain:
    UPN:
[3] Logon session 00000000:000003e5:
    User name:    NT AUTHORITY\LOCAL SERVICE
    Auth package: Negotiate
    Logon type:   Service
    Session:      0
    Sid:          S-1-5-19
    Logon time:   2012-01-16 22:03:40
    Logon server:
    DNS Domain:
    UPN:
[4] Logon session 00000000:00021ed2:
    User name:    NT AUTHORITY\ANONYMOUS LOGON
    Auth package: NTLM
    Logon type:   Network
    Session:      0
    Sid:          S-1-5-7
    Logon time:   2012-01-16 22:03:46
    Logon server:
    DNS Domain:
    UPN:
[5] Logon session 00000000:000882c2:
    User name:    LAPT8\jeh
    Auth package: NTLM
    Logon type:   Interactive
    Session:      1
    Sid:          S-1-5-21-1488595123-1430011218-1163345924-1000
    Logon time:   2012-01-17 01:34:46
    Logon server: LAPT8
    DNS Domain:
    UPN:
[6] Logon session 00000000:000882e3:
    User name:    LAPT8\jeh
    Auth package: NTLM
    Logon type:   Interactive
    Session:      1
    Sid:          S-1-5-21-1488595123-1430011218-1163345924-1000
    Logon time:   2012-01-17 01:34:46
    Logon server: LAPT8
    DNS Domain:
    UPN:
```

Информация, представленная для сеанса, включает SID и имя пользователя, связанного с сеансом, а также пакет аутентификации сеанса и время входа в систему. Учтите, что пакет аутентификации Negotiate, показанный в сеансе входа в систему 2 в предыдущем выводе, будет пытаться провести аутентификацию с использованием Kerberos или NTLM, в зависимости от того, какой из них окажется наиболее подходящим для запроса на аутентификацию.

LUID сеанса выводится в строке «Logon Session» каждого блока сеанса, и с помощью утилиты Handle (также из набора Sysinternals) вы можете найти маркеры, представляющие конкретный сеанс входа в систему. Например, чтобы найти маркеры для сеанса входа в систему 5 в только что показанном примере вывода, вы можете ввести следующую команду:

```
C:\Windows\system32>handle -a 882c2
Handle v3.46
Copyright (C) 1997-2011 Mark Russinovich
Sysinternals - www.sysinternals.com
System          pid: 4    type: Directory    D60: \Sessions\0\DosDevices\00000000-
000882c2
winlogon.exe       pid: 440    type: Event           DC:
 \BaseNamedObjects\00000000000882c2_WlballoonSmartCardUnlockNotificationEventName
winlogon.exe       pid: 440    type: Event           E4:
 \BaseNamedObjects\00000000000882c2_WlballoonKerberosNotificationEventName
winlogon.exe       pid: 440    type: Event          1D4:
 \BaseNamedObjects\00000000000882c2_WlballoonAlternateCredsNotificationEventName
lsass.exe          pid: 492    type: Token           508: LAPT8\jeh:882c2
lsass.exe          pid: 492    type: Token           634: LAPT8\jeh:882c2
svchost.exe        pid: 892    type: Token           7C4: LAPT8\jeh:882c2
svchost.exe        pid: 960    type: Token           E70: LAPT8\jeh:882c2
svchost.exe        pid: 960    type: Token          1034: LAPT8\jeh:882c2
svchost.exe        pid: 960    type: Token          1194: LAPT8\jeh:882c2
svchost.exe        pid: 960    type: Token          1384: LAPT8\jeh:882c2
```

Затем Winlogon ищет в реестре значение параметра HKLM\SOFTWARE\Microsoft\Windows NT\Current Version\Winlogon\Userinit и создает процесс для запуска в зависимости от содержимого хранящегося там строкового значения. (Это значение может включать несколько имен файлов с расширениями .EXE, отделенных друг от друга запятыми.) Значением по умолчанию является процесс Userinit.exe, загружающий профиль пользователя, а затем создающий процесс для запуска того, что указано в значении параметра HKCU\SOFTWARE\Microsoft\Windows NT\Current Version\Winlogon\Shell, если этот параметр существует. Изначально он отсутствует. Если он отсутствует, Userinit.exe делает то же самое для параметра HKLM\SOFTWARE\Microsoft\Windows NT\Current Version\Winlogon\Shell, в значении которого изначально находится Explorer.exe. Затем происходит выход из Userinit (именно поэтому для Explorer.exe не показывается родительский процесс при его просмотре в Process Explorer).

Гарантированная аутентификация

Основная проблема, связанная с аутентификацией на основе пароля, заключается в том, что пароль может быть вскрыт или украден и использован третьими

лицами для нанесения вреда. Новым в Windows 7 и Windows Server 2008/R2 стал механизм, отслеживающий ту защищенность, с которой пользователь прошел аутентификацию пользователя. Это позволяет объектам быть защищенными от доступа, если пользователь не прошел аутентификацию безопасным образом. Аутентификация с помощью смарткарт считается более защищенной формой аутентификации по сравнению с парольной аутентификацией.

На системах, присоединенных к домену, администратор домена может указать отображение между идентификатором объекта — Object Identifier (OID), — являющимся уникальной цифровой строкой, представляющей определенный тип объекта, сертификатом, использующимся для аутентификации пользователя (например, на смарткарте или на аппаратном маркере безопасности), и идентификатором безопасности (SID), помещаемым в пользовательский маркер доступа, при успешной аутентификации пользователя системой. ACE-элемент в DACL объекта может указать такой SID в качестве части пользовательского маркера с целью получения пользователем доступа к объекту. С технической точки зрения это называется групповым требованием (group claim). Иными словами, пользователь требует участия в конкретной группе, что позволяет получить конкретные права доступа к конкретным объектам, с требованием, основанным на механизме аутентификации. Это свойство изначально не включено, и оно должно быть настроено администратором домена в том домене, в котором используется аутентификация на основе сертификата.

Гарантированная аутентификация строится на основе существующих в Windows возможностей обеспечения безопасности таким образом, чтобы предоставить IT-администраторам и всем заинтересованным в IT-безопасности предприятия максимальную гибкость. Предприятия решают, какие OID-идентификаторы встраивать в сертификаты, используемые ими для аутентификации и для отображения конкретных OID на универсальные группы Active Directory (SID-идентификаторы). Принадлежность пользователя к группе может использоваться для идентификации того, был ли использован сертификат в ходе операции входа в систему. Различные сертификаты могут иметь различные политики изданий и, таким образом, различные уровни безопасности, которые могут использоваться для защиты особо важных объектов (например, файлов или чего-нибудь еще, что может иметь дескриптор безопасности).

Протоколы аутентификации — authentication protocols (AP) — получают OID-идентификаторы от сертификатов в ходе аутентификации, основанной на использовании сертификатов. Эти OID-идентификаторы должны отображаться на SID-идентификаторы, которые, в свою очередь, обрабатываются в ходе расширения участия в группе и помещаются в маркер доступа. Отображение OID на универсальную группу указывается в Active Directory.

В качестве примера, у организации может быть несколько политик выпуска сертификатов с именами Contractor (подрядчик), Full Time Employee (штатный работник) и Senior Management (представитель высшего руководства), что отображается соответственно на универсальные группы Contractor-Users, FTE-Users и SM-Users. У пользователя Abby есть смарткарта с сертификатом, выпущенным с использованием политики выпуска Senior Management, и когда она входит в систему, используя свою смарткарту, она получает дополнительное членство в группе (что представлено с помощью SID в ее маркере доступа), показывающее, что она входит в группу SM-Users. Права доступа могут быть установлены (с ис-

пользованием ACL) на такие объекты, доступ к которым предоставляется только представителям групп FTE-Users или SM-Users (идентифицируемым по их SID внутри ACE). Если Abby входит в систему с помощью своей смарткарты, она может получать доступ к таким объектам, но если она входит в систему, используя только свое имя пользователя и пароль (не пользуясь смарткартой), она не может получить доступ к таким объектам, поскольку в ее маркере доступа у нее не будет членства ни в группе FTE-Users, ни в группе SM-Users. Пользователь Toby, вошедший в систему с помощью смарткарты, имеющей сертификат, выпущенный с использованием политики выпуска Contractor, не сможет получить доступ к объекту, имеющему ACE, который требует членства в группе FTE-Users или в группе SM-Users.

Биометрическая среда для аутентификации пользователей

Windows предоставляет стандартизированный механизм для поддержки определенных типов биометрических устройств, в частности сканеров отпечатка пальца, чтобы обеспечивать идентификацию пользователей. Подобно многим другим таким средам, Windows Biometric Framework была разработана для изолирования различных функций, используемых в поддержке подобных устройств таким образом, чтобы минимизировать код, необходимый для реализации нового устройства.

Основные компоненты среды Windows Biometric Framework показаны на рис. 6.14. Кроме особо оговоренных в следующем списке, все эти компоненты поставляются вместе с Windows:

- **Биометрическая служба Windows — Windows Biometric Service (%SystemRoot%\System32\Wbiosrvc.dll).** Предоставляет среду выполнения процесса, в которой может выполняться один или несколько поставщиков биометрических служб.
- **Биометрический API Windows — Windows Biometric API.** Позволяет существующим компонентам Windows, например WinLogon и LoginUI, получать доступ к биометрической службе. Приложения сторонних разработчиков имеют доступ к биометрическим API-функциям и могут использовать биометрический сканер для функций, отличающихся от входа в систему Windows. Примером функции в этом API может послужить WinBioEnumServiceProviders. Биометрический API-интерфейс представлен в библиотеке %SystemRoot%\System32\Winbio.dll.
- **Поставщик биометрической службы отпечатков пальцев — Fingerprint Biometric Service Provider.** Служит оболочкой для функций биометрических адаптеров того или иного типа с целью присутствия общего интерфейса, независимого от типа биометрии, с биометрической службой Windows Biometric Service. В будущем с помощью дополнительных поставщиков биометрической службы могут быть поддержаны дополнительные типы биометрии, например сканирование радужной оболочки глаз или анализ индивидуальных особенностей голоса. Поставщик биометрической службы, в свою очередь, использует три адаптера, которые являются DLL-библиотеками пользовательского режима:
  - Адаптер сенсора, предоставляющий функциональные возможности сканера, связанные с получением данных. Адаптер сенсора будет, как правило, исполь-

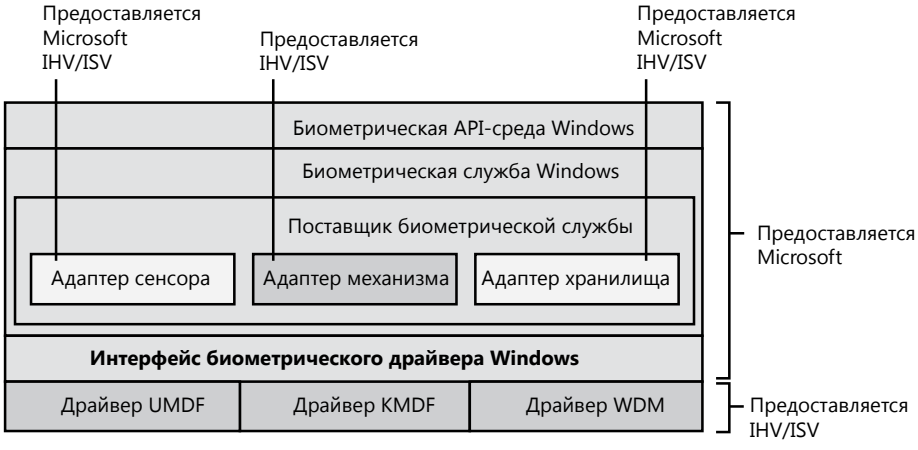


Рис. 6.14. Компоненты и архитектура среды Windows Biometric Framework

зовать для доступа к оборудованию сканера Windows-вызовы ввода-вывода. Windows предоставляет адаптер сенсора, который может использоваться с простыми сенсорами, для которых имеется драйвер Windows Biometric Device Interface (WBDI). Для более сложных сенсоров адаптеры пишутся их поставщиками.

- Адаптер механизма, предоставляющий функциональные возможности по обработке и сравнению, характерной для формата исходных данных сканера и других свойств. Фактическая обработка и сравнение могут проводиться внутри DLL-библиотеки адаптера механизма, или DLL может обмениваться данными с каким-нибудь другим модулем. Адаптер механизма всегда предоставляется поставщиком сенсора.
- Адаптер хранилища предоставляет набор функций безопасного хранения. Они используются для хранения и извлечения шаблонов, которые используются адаптером механизма для сравнения с данными сканера биометрических данных. Windows предоставляет адаптер хранилища, использующий криптографические службы Windows, и стандартное дисковое хранилище файлов. Поставщик сенсора может предоставлять другой адаптер хранилища.

- **Интерфейс биометрического драйвера Windows — Windows Biometric Driver Interface.** Это набор интерфейсных определений (основные функциональные коды IRP, коды DeviceIoControl и т. д.), которым должен соответствовать любой драйвер для устройства биометрического сканирования, если ему нужно быть совместимым с биометрической службой Windows. Описание WBDI дано в документации по Windows Driver Kit. В этот же инструментарий включен пример WBDI-драйвера.
- **Функциональный драйвер для реального устройства биометрического сканирования.** Предоставляет самый последний WBDI и обычно для доступа к устройству сканирования использует службы низкоуровневого драйвера шины, например драйвера шины USB. Это может быть драйвер среды драйверов пользовательского режима — User-Mode Driver Framework (UMDF), драйвер среды драйверов режима ядра — Kernel-Mode Driver Framework

(KMDF), или драйвер модели драйверов Windows — Windows Driver Model (WDM). Этот драйвер всегда предоставляется поставщиком сенсора. Microsoft рекомендует использовать для сканера UMDF и аппаратный интерфейс USB.

Обычная последовательность операций поддержки входа в систему посредством сканирования отпечатка пальца может быть следующей:

1. После инициализации адаптер сенсора получает от поставщика службы запрос на получение данных. Адаптер сенсора, в свою очередь, отправляет запрос DeviceIoControl с кодом управления IOCTL_BIOMETRIC_CAPTURE_DATA драйверу WBDI для устройства сканирования отпечатка пальца.
2. Драйвер WBDI переводит сканер в режим захвата и ставит в очередь запрос IOCTL_BIOMETRIC_CAPTURE_DATA до тех пор, пока не произойдет сканирование отпечатка.
3. Потенциальный пользователь проводить пальцем по сканеру. Драйвер WBDI получает об этом уведомление, получает необработанные данные сканирования от сенсора и возвращает эти данные драйверу сенсора в буфере, связанном с запросом IOCTL_BIOMETRIC_CAPTURE_DATA.
4. Адаптер сенсора предоставляет данные поставщику биометрической службы отпечатков пальцев — Fingerprint Biometric Service Provider, — который, в свою очередь, передает данные адаптеру механизма.
5. Адаптер механизма обрабатывает исходные данные, придавая им форму, совместимую с его хранилищем образцов.
6. Поставщик биометрической службы отпечатков пальцев использует адаптер хранилища для получения образцов и соответствующих идентификаторов безопасности от хранилища обеспечения безопасности. Он вызывает адаптер механизма для сравнения каждого образца с обработанными данными сканирования. Адаптер механизма возвращает статус, показывающий, найдено соответствие или нет.
7. Если соответствие найдено, биометрическая служба уведомляет процесс WinLogon через DLL поставщика учетных данных об успешном входе и передает ему идентификатор безопасности распознанного пользователя. Это уведомление отправляется через сообщение усовершенствованного вызова локальных процедур — Advanced Local Procedure Call, — предоставляя путь, который не может быть подделан.

Управление учетными записями пользователей и виртуализация

Система UAC предназначена для того, чтобы позволить пользователям работать с правами обычного пользователя, в отличие от работы с правами администратора. Без административных прав пользователи не могут случайно (или преднамеренно) изменить настройки системы, вредоносные программы не смогут обычным образом изменить настройки безопасности системы или отключить антивирусное программное обеспечение, и пользователи не смогут получить несанкционированный доступ к конфиденциальной информации других пользо-

вателей на общих компьютерах. Работа с правами обычного пользователя может таким образом подавить воздействие вредоносных программ и защитить важные данные на общих компьютерах.

Чтобы ввести в практику пользователя работу под обычной учетной записью, при разработке UAC пришлось решить ряд проблем. Во-первых, поскольку модель использования Windows предполагала наделение административными правами, разработчики программного обеспечения предполагали, что их программы будут запускаться с этими правами и смогут тем самым получать доступ к любому файлу, к разделу реестра или к настройкам операционной системы и вносить в них изменения. Во-вторых, при разработке UAC пришлось решать проблему, связанную с тем, что пользователям иногда нужны административные права для выполнения таких операций, как установка программного обеспечения, изменение системного времени и открытие портов в брандмауэре.

В UAC эти проблемы были решены таким образом, что большинство приложений запускалось с правами обычного пользователя, даже если пользователь вошел в систему под учетной записью с административными правами. Но в то же время UAC позволяла обычным пользователям по мере надобности получать доступ к административным правам, когда они требовались устаревшим приложениям или когда нужно было изменить конкретные настройки системы.

Как уже ранее упоминалось, UAC выполняет эти задачи с помощью создания фильтрованных маркеров администратора, а также с помощью обычных маркеров администратора, когда пользователь вошел в систему под административной учетной записью. Ко всем процессам, создаваемым в ходе сеанса пользователя, будут, как правило, применяться фильтрованные маркеры администратора, чтобы те приложения, которые могут выполняться с правами обычного пользователя, именно так и выполнялись. Но пользователи с правами администратора могут запустить программу или выполнить другие функции, требующие полных административных прав путем повышения привилегий в системе UAC.

Windows также позволяет определенным задачам, которые ранее считались предназначенными для администраторов, выполняться обычными пользователями, повышая тем самым удобства использования обычной среды пользователя. Например, существуют такие настройки групповой политики, которые могут позволить обычным пользователям устанавливать драйверы принтеров или других устройств, одобренные IT-администраторами, и устанавливать элементы управления ActiveX с одобренных администраторами сайтов.

И наконец, когда разработчики программного обеспечения тестируют его в среде UAC, они ориентируются на разработку приложений, которые могут работать без административных прав. По сути, неадминистративным программам не требуется работа с привилегиями администратора, а программы, которым часто требуются привилегии администратора, являются, как правило, устаревшими, использующими старые API-функции или технологии, и такие программы должны быть обновлены.

Вместе взятые все эти изменения устраняют для пользователей необходимость постоянной работы с административными правами.

Файловая система и виртуализация реестра

Хотя некоторые устаревшие программы требуют административных прав, многим программам не нужно хранить пользовательские данные в глобальных для систе-

мы местах. При выполнении приложения оно может запускаться под разными учетными записями пользователя и поэтому должно хранить данные конкретного пользователя в существующем для каждого пользователя каталоге %AppData%, а также сохранять настройки каждого пользователя в находящемся в реестре профиле пользователя в разделе HKEY_CURRENT_USER\Software. Учетные записи обычных пользователей не имеют прав доступа по записи к каталогу %ProgramFiles% или к разделу HKEY_LOCAL_MACHINE\Software, но поскольку большинство систем Windows являются однопользовательскими и большинство пользователей до реализации UAC были администраторами, приложения, которые некорректно сохраняли пользовательские данные и настройки в этих местах, все равно работали.

Windows позволяет таким устаревшим приложениям работать под учетными записями обычных пользователей благодаря виртуализации файловой системы и пространства имен реестра. Когда приложение изменяет системное глобальное местоположение в файловой системе или в реестре и эта операция дает сбой по причине запрета доступа, Windows перенаправляет операцию в область, предназначенную для конкретного пользователя. Когда приложение производит чтение из системного глобального места, Windows сначала проверяет наличие данных в области конкретного пользователя, если они не будут найдены, позволяет предпринять попытку чтения из глобального места.

Windows всегда будет разрешать такой тип виртуализации, за исключением следующих случаев:

- Приложение является 64-разрядным. Поскольку виртуализация является исключительно технологией обеспечения совместимости приложений, предназначенной для содействия работе устаревших программ, ее применение возможно только на 32-разрядных приложениях. Мир 64-разрядных приложений является относительно новым, и разработчики должны следовать установкам проектирования по созданию приложений, совместимых с их использованием обычными пользователями.
- Приложение уже работает с административными правами. В таком случае необходимость в виртуализации отпадает.
- Операция инициирована вызывающей процедурой режима ядра.
- Операция выполняется во время заимствования прав вызывающим процессом. Например, любые операции, не инициированные процессом, классифицированным в соответствии с его определением как устаревший, включая сетевые доступы к общим файлам, виртуализации не подвергаются.
- Исполняемый образ для процесса имеет совместимый с UAC манифест (определенный настройкой requestedExecutionLevel, рассматриваемой в следующем разделе).
- Администратор не имеет прав доступа по записи к файлу или к разделу реестра. Своим существованием это исключение обязано навязыванию принципа обратной совместимости, поскольку устаревшие приложения получили бы отказ в выполнении еще до реализации UAC, даже если приложение было запущено с правами администратора.
- Виртуализация никогда не применяется в отношении служб.

Статус виртуализации процесса (как уже ранее упоминалось, этот статус хранится в маркере в виде флага) можно увидеть путем добавления к странице процессов Диспетчера задач столбца Виртуализация UAC (UAC Virtualization), как показано на рис. 6.15. Многие компоненты Windows, включая Диспетчер окон рабочего стола — Desktop Window Manager (Dwm.exe), Клиент-серверную подсистему времени выполнения — Client Server Run-Time Subsystem (Csrss.exe) и Explorer, работают с отключенной виртуализацией, поскольку у них имеется совместимый с UAC манифест, или они запущены с административными правами и поэтому не допускают виртуализацию. У Internet Explorer (Iexplore.exe) виртуализация включена, поскольку он может владеть несколькими элементами управления ActiveX и сценариями и должен допускать, что они не создавались для корректной работы с правами обычных пользователей.

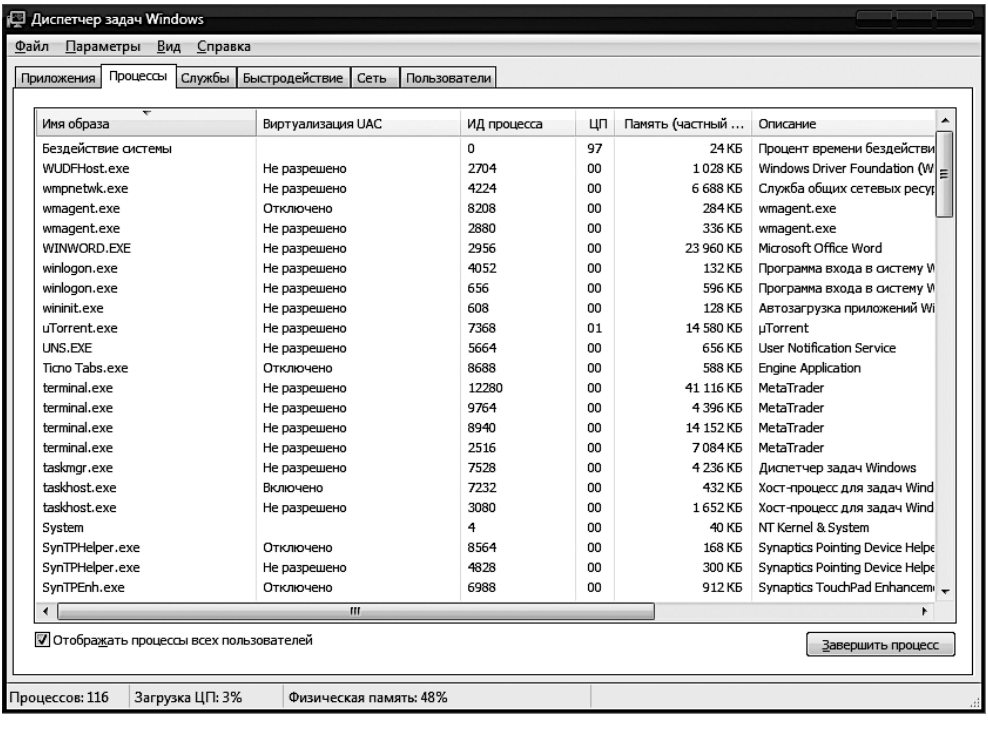


Рис. 6.15. Использование Диспетчера задач для просмотра статуса виртуализации

Кроме виртуализации файловой системы и реестра, некоторым приложениям требуется дополнительная помощь для нормальной работы с правами обычных пользователей. Например, приложение, тестирующее учетную запись, под которой оно было запущено на членство в группе Администраторы, могло бы работать, но не будет, если учетная запись не входит в эту группу. В Windows определен ряд прокладок обеспечения совместимости приложений, чтобы все равно позволить этим приложениям работать. Прокладки чаще всего применяются к устаревшим приложениям для работы с правами обычных пользователей, как показано в табл. 6.10. Учтите, что по мере надобности с помощью настроек

локальной политики безопасности виртуализация может быть полностью отключена.

Таблица 6.10. Прокладки виртуализации UAC

Флаг	Значение
ElevateCreateProcess	Изменяет CreateProcess для обработки ошибок запрошенного повышения привилегий — ERROR_ELEVATION_REQUIRED путем вызова службы информации приложений (application information service) для вывода приглашения на запрос повышения привилегий
ForceAdminAccess	Служит прокладкой для участников группы Администраторы
VirtualizeDeleteFile	Служит прокладкой для успешного удаления глобальных файлов и каталогов
LocalMappedObject	Заставляет объекты глобальных разделов отображаться на пользовательское пространство имен
VirtualizeHKCRLite	Перенаправляет глобальную регистрацию COM-объектов в место, определяемое для каждого пользователя
VirtualizeRegisterTypeLib	Превращает помашинную typelib-регистрацию в регистрации для каждого пользователя

Файловая виртуализация

К местам файловой системы, виртуализированным для устаревших процессов, относятся %ProgramFiles%, %ProgramData% и %SystemRoot%, за исключением некоторых конкретных подкаталогов. Но любой файл с расширением, указывающим на его исполняемую природу, включая .exe, .bat, .scr, .vbs и др., из виртуализации исключается. Это означает, что программы, обновляющие сами себя, будут под обычной учетной записью получать отказ в выполнении вместо создания закрытых версий своих исполняемых файлов, невидимых администратору, работающему в режиме глобального обновления.

Изменения виртуализированных каталогов, вносимые устаревшими процессами, перенаправляются в виртуальный корневой каталог пользователя %LocalAppData%\VirtualStore. Компонент Local в этом пути подчеркивает тот факт, что виртуализированные файлы не подвергаются роумингу (roam) со всем остальным профилем, когда у учетной записи есть блуждающий профиль (roaming profile). Если в Explorer перейти к каталогу, содержащему виртуализированные файлы, он показывает на панели инструментов кнопку с надписью Файлы совместимости (Compatibility Files), как показано на рис. 6.16. После щелчка на этой кнопке происходит переход в соответствующий подкаталог VirtualStore для показа виртуализированных файлов.

Виртуализация файловой системы реализуется драйвером фильтра виртуализации UAC — UAC File Virtualization Filter Driver (%SystemRoot%\System32\Drivers\Luafv.sys). Поскольку это драйвер фильтра файловой системы, он видит все локальные операции файловой системы, но реализует свою функциональность только для операций с устаревших процессов. Как показано на рис. 6.17,

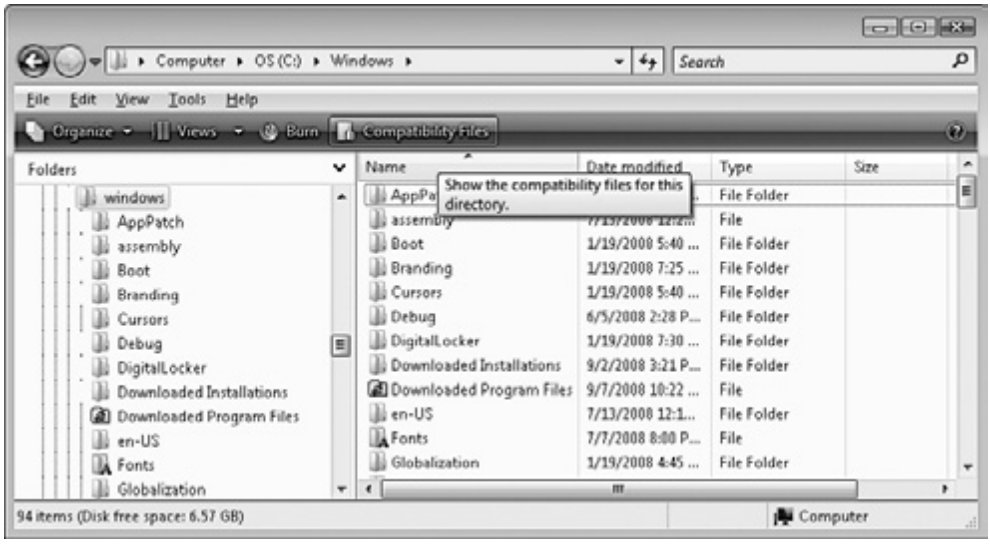


Рис. 6.16. Виртуализированные файлы отображаются здесь

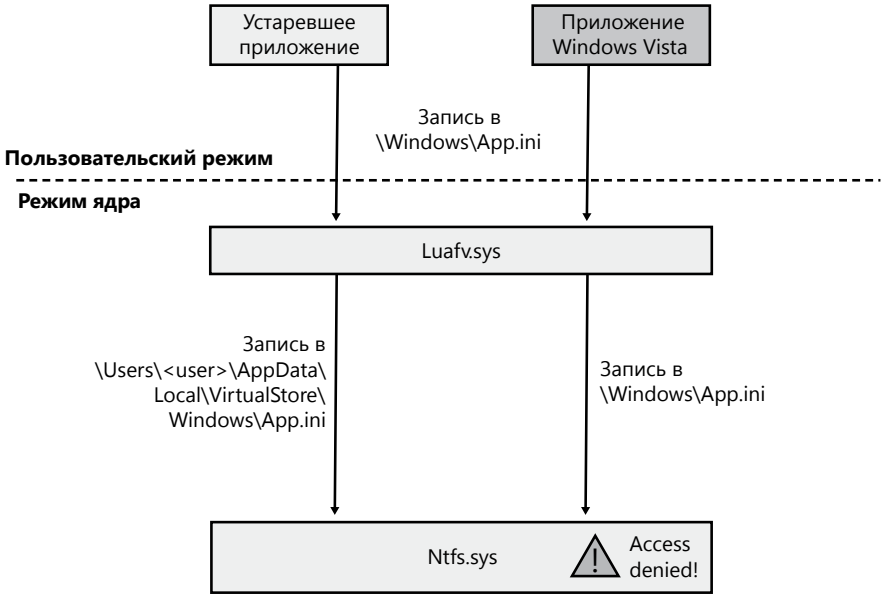


Рис. 6.17. Действие драйвера фильтра виртуализации файлов системы UAC

драйвер фильтра изменяет путь целевого файла для устаревшего процесса, который создает файл в глобальном системном месте, но не изменяет его для невиртуализированных процессов с правами обычных пользователей. Исходные полномочия в отношении каталога \Windows запрещают доступ к приложениям, написанным с поддержкой UAC, но устаревшие процессы действуют таким

образом, будто операция прошла успешно, когда на самом деле файл создается в том месте, которое полностью доступно пользователю.

ПРИМЕЧАНИЕ

Чтобы добавить дополнительные расширения к списку исключений, введите их в параметр реестра HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Luafv\Parameters\ExcludedExtensionsAdd и перезапустите систему. Для отделения нескольких расширений друг от друга воспользуйтесь мультистроковым типом параметра и не включайте в имя расширения лидирующую точку.

ЭКСПЕРИМЕНТ: ПОВЕДЕНИЕ, СВЯЗАННОЕ С ВИРТУАЛИЗАЦИЕЙ ФАЙЛОВ

В данном эксперименте мы будет включать и отключать виртуализацию окна командной строки и наблюдать за примерами поведения для демонстрации виртуализации файлов системой UAC:

1. Откройте окно командной строки без повышенных привилегий (для этой работы у вас должна быть включена система UAC) и включите для него виртуализацию. Статус виртуализации процесса можно изменить, выбрав пункт Виртуализация UAC (UAC Virtualization) в меню быстрого вызова, появляющегося при щелчке правой кнопкой мыши на имени процесса в Диспетчере задач.
2. Перейдите в каталог C:\Windows и воспользуйтесь для записи файла следующей командой:

```
echo hello-1 > test.txt
```

3. Теперь выведите список содержимого каталога:

```
dir test.txt
```

   Вы увидите, что файл появился.

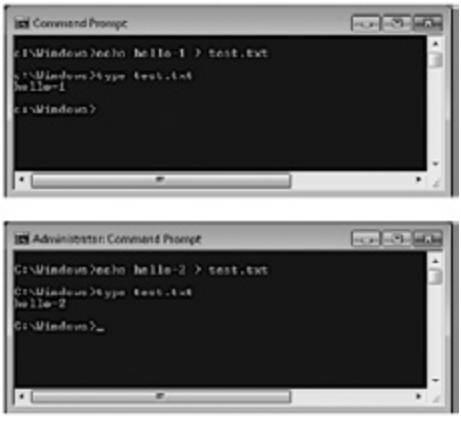
4. Теперь отключите виртуализацию, щелкнув правой кнопкой на имени процесса на странице Процессы (Processes) в Диспетчере задач, и отмените выбор Виртуализация UAC (UAC Virtualization), а затем выведите список содержимого каталога, повторив действие пункта 3. Обратите внимание на то, что файл исчез. Но вывод содержимого каталога VirtualStore покажет наличие файла:

```
dir %LOCALAPPDATA%\VirtualStore\Windows\test.txt
```

5. Опять включите виртуализацию для этого процесса.
6. Чтобы посмотреть на более сложный сценарий, создайте новое окно командной строки, но теперь с повышенными привилегиями, а затем повторите действия пункта 2 и 3, используя строку «hello-2».
7. Изучите текст внутри этих файлов, используя следующую команду в обоих окнах командной строки:

```
echo test.txt
```

Ожидаемый вывод показан в следующих двух рисунках.



8. И наконец, из своего окна командной строки с повышенными привилегиями удалите файл test.txt:

```
del test.txt
```

Повторите действие из пункта 6 эксперимента. Обратите внимание на то, что в окне командной строки с повышенными привилегиями найти файл уже не удается, а вот в окне командной строки обычного пользователя опять показано старое содержимое файла. В этом эксперименте демонстрируется рассмотренный ранее механизм обхода отказов — операции чтения сначала будут смотреть в виртуальное место хранения, создаваемое для каждого пользователя, но если файл отсутствует, будет предоставлен доступ по чтению к системному месту хранения. ■

Виртуализация реестра

Виртуализация реестра реализована несколько иначе, чем виртуализация файловой системы. Виртуализированные разделы реестра включают большую часть ветви HKEY_LOCAL_MACHINE\Software, но есть и ряд исключений:

- HKLM\Software\Microsoft\Windows
- HKLM\Software\Microsoft\Windows NT
- HKLM\Software\Classes

Виртуализации подвергаются только те разделы, которые обычно изменяются устаревшими приложениями, но при этом не создают проблем совместимости или взаимодействия. Windows перенаправляет изменения виртуализированных разделов, вносимые устаревшими приложениями в создаваемый в реестре виртуальный корневой раздел пользователя HKEY_ CURRENT_USER\Software\Classes\VirtualStore. Раздел находится в пользовательском кусте Classes, %LocalAppData%\Microsoft\Windows\UsrClass.dat, который, подобно любым другим виртуализированным файловым данным, не подвергается роумингу вместе с блуждающим профилем пользователя. Вместо ведения постоянного списка виртуализированных мест, как это делается Windows для файловой системы, статус виртуализации раздела сохраняется в виде комбинации флагов, показанных в табл. 6.11.

Таблица 6.11. Флаги виртуализации реестра

Флаг	Значение
REG_KEY_DONT_VIRTUALIZE	Указывает, разрешена ли виртуализация для этого раздела. Если флаг установлен, виртуализация отключена
REG_KEY_DONT_SILENT_FAIL	Если флаг REG_KEY_DONT_VIRTUALIZE установлен (виртуализация отключена), этот флаг указывает, что устаревшее приложение, которое не сможет получить доступ при выполнении операции над разделом, вместо этого взамен прав, запрошенных приложением, получает в отношении раздела права максимального разрешения MAXIMUM_ALLOWED (любого доступа, предоставленного учетной записи). Если этот флаг установлен, он также косвенно отключает виртуализацию
REG_KEY_RECURSE_FLAG	Определяет, будут ли флаги виртуализации распространяться на дочерние разделы (подразделы) этого раздела

Для вывода текущего состояния виртуализации раздела или для установки этого состояния можно воспользоваться включенной в состав Windows утилитой Reg.exe с ключом flags. Обратите внимание, что на рис. 6.18 показано, что раздел HKLM\Software полностью виртуализирован, а подраздел Windows (и все его дочерние подразделы) имеют только установленный флаг безмолвной ошибки REG_KEY_DONT_SILENT_FAIL.

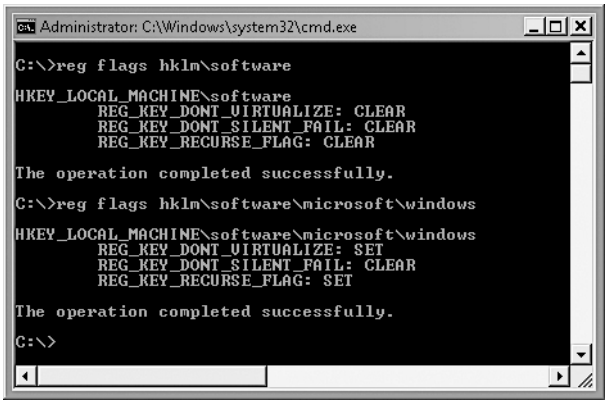


Рис. 6.18. Используемые UAC флаги виртуализации разделов Software и Windows

В отличие от файловой виртуализации, использующей драйвер фильтра, реестровая виртуализация реализована в диспетчере конфигурации (см. главу 4). Как и в виртуализации файловой системы, устаревший процесс, создающий подраздел виртуализированного раздела, перенаправляется на пользовательский виртуальный корневой раздел реестра, но UAC-совместимый процесс получить доступ с исходными полномочиями не сможет. Это показано на рис. 6.19.

Повышение привилегий

Даже если пользователи запускают только программы, совместимые с правами обычного пользователя, некоторые операции все же требуют административных

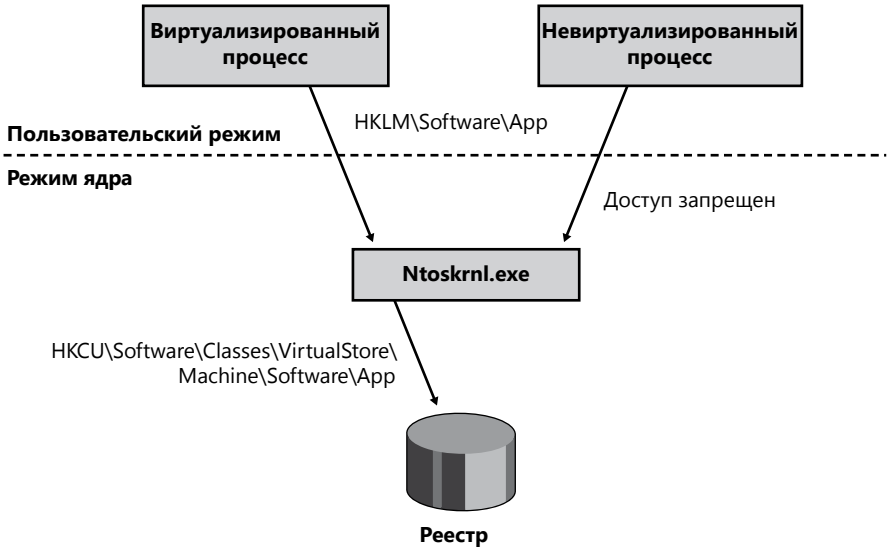


Рис. 6.19. Используемая UAC операция виртуализации реестра

прав. Например, подавляющее большинство установок программного обеспечения требуют административных прав для создания каталогов и разделов реестра в глобальных местах системы или установки служб или драйверов устройств. Изменение системных глобальных настроек Windows и приложений также требует административных прав, то же самое относится и к функции родительского контроля. Можно было бы выполнить большинство этих операций путем переключения на специальную административную учетную запись, Но связанные с этим неудобства, скорее всего, приведут к тому, что большинство пользователей для выполнения своих повседневных задач, основная часть которых не требует административных прав, останутся в административной учетной записи.

Важно знать, что повышение привилегий в системе UAC является удобством, а не границами безопасности. Граница безопасности требует, чтобы политика безопасности однозначно определяла, что именно может проходить через границу. Примером границы безопасности в Windows являются учетные записи пользователей, поскольку один пользователь не может получить доступ к данным, принадлежащим другому пользователю, не имея разрешения этого пользователя.

Поскольку повышение привилегий не является границами безопасности, нельзя дать гарантий, что вредоносная программа, выполняемая в системе с правами обычного пользователя, не может скомпрометировать процесс с повышенными привилегиями для получения административных прав. Например, диалоговые окна получения повышенных привилегий всего лишь идентифицируют исполняемый код, чьи привилегии будут повышены, и ничего не говорят о том, чем он будет заниматься при своем выполнении.

Выполнение с административными правами

В Windows включена усовершенствованная функциональная возможность «run as» (выполнение от имени), чтобы обычные пользователи могли, не испытывая

неудобств, запускать процессы с административными правами. Эта функциональная возможность требует предоставления приложениям способа идентификации операций, для которых система, если это необходимо, может получить административные права от имени приложения. (К этой теме мы еще вскоре вернемся.)

Чтобы позволить пользователям временно выполнять обязанности системного администратора с правами обычного пользователя и чтобы при этом не приходилось вводить пользовательские имена и пароли при каждой необходимости получения административных прав, в Windows применяется механизм, который называется режимом, одобренным администратором — Admin Approval Mode (AAM). Это свойство при входе в систему создает две идентичности: одну с правами обычного пользователя и другую с административными правами. Поскольку каждый пользователь в системе Windows является либо обычным пользователем, либо действует большей частью как обычный пользователь в AAM, разработчики должны предполагать, что все пользователи Windows являются обычными пользователями, что приведет к увеличению количества программ, работающих с правами обычного пользователя без виртуализации или прокладок.

Предоставление административных прав процессу называется повышением привилегий. Когда повышение привилегий выполняется под учетной записью обычного пользователя (или пользователя, входящего в административную группу, но не в группу Администраторы), это называется повышением через плечо — over-the-shoulder (OTS), потому что оно требует ввода учетных данных для учетной записи, входящей в группу Администраторы, что напоминает заполнение данных пользователем, набирающим текст через плечо обычного пользователя. Повышение привилегий, выполняемое AAM-пользователем, называется согласованным повышением, потому что пользователь просто должен одобрить назначение его административных прав.

Автономные системы, которые обычно представлены домашними компьютерами и объединенными в домен системами, рассматривают AAM-доступ со стороны удаленных пользователей по-другому, потому что подключенные к домену компьютеры в своих полномочиях на ресурсы могут использовать доменные административные группы. Когда пользователь обращается к общему файлу автономного компьютера, Windows запрашивает принадлежащую удаленному пользователю идентичность обычного пользователя, но на системах, объединенных в домен, Windows принимает во внимание все пользовательские участия в доменной группе, запрашивая административную идентичность пользователя.

Исполнение образа, запрашивающего административные права, вызывает службу сведений о приложении — application information service (AIS, содержится в файле %SystemRoot%\System32\Appinfo.dll), которая запускает внутри службы хост-процесс (%SystemRoot%\System32\Svchost.exe) для запуска Consent.exe (%SystemRoot%\System32\Consent.exe). Consent захватывает растровое изображение экрана, применяет к нему эффект затемнения, переключается на рабочий стол, доступный только учетной записи локальной системы (защищенный рабочий стол), вырисовывает растровое изображение в качестве фона и отображает диалоговое окно повышения привилегий, содержащее сведения об исполняемом файле. Вывод этого диалогового окна в отдельном рабочем столе не дает любому приложению, запущенному под учетной записью пользователя, изменять внешний вид диалогового окна.

Если образ является компонентом Windows, имеющим цифровую подпись Microsoft, и этот образ находится в системном каталоге Windows, в верхней части диалогового окна отображается синяя полоска, показанная в верхнем фрагменте рис. 6.20, с синим и золотистым щитом, находящимся слева на этой полосе. Если образ подписан не Microsoft, а какой-нибудь другой компанией или он подписан Microsoft, но находится не в дереве каталогов Windows, а в другом месте, щит становится просто синим и на нем появляется знак вопроса. Если образ не подписан, то и поле щита и полоса становятся оранжевыми, и на щите появляется восклицательный знак, и в сообщении подчеркивается, что издатель образа неизвестен. В диалоговом окне повышения привилегий показывается значок образа, описание и издатель для образов, имеющих цифровую подпись, но для образов без подписи показывается только имя файла и надпись «Издатель: Неизвестно» («Unknown publisher»). Такие различия затрудняют маскировку вредоносных программ под законное программное обеспечение. Кнопка Показать подробности (Show details) в нижней части диалогового окна при щелчке на ней расширяет окно, чтобы показать командную строку, которая будет передана исполняющей системе при запуске образа на выполнение.

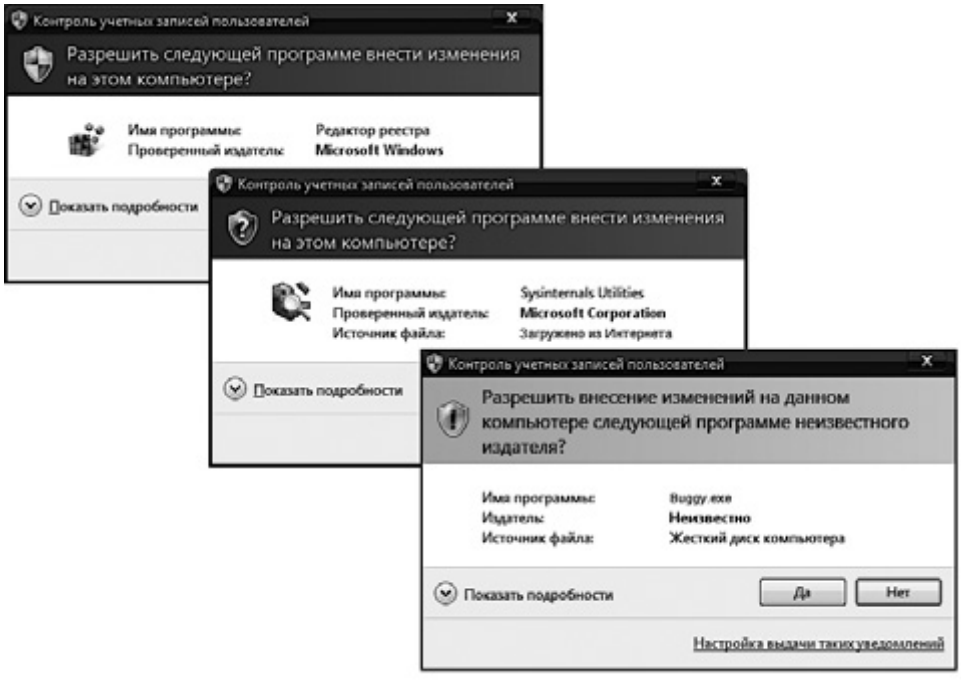


Рис. 6.20. Диалоговые окна повышения привилегий ААС UAC, выводимые на основе состояния цифровой подписи образа

Диалоговое окно OTS-согласия, показанное на рис. 6.21, имеет похожий вид, но содержит приглашение на ввод учетных данных администратора. В нем будет выводиться список любых учетных записей с административными правами.

Рис. 6.21. Диалоговое окно OTS-согласия

Если пользователь отклоняет повышение привилегий, Windows возвращает процессу, инициировавшему запуск ошибку отказа в доступе. Когда пользователь соглашается с повышением привилегий либо путем ввода учетных данных администратора, либо путем щелчка на кнопке Да (Yes), AIS вызывает функцию CreateProcessAsUser для запуска процесса с соответствующей административной идентичностью. Хотя AIS с технической точки зрения является родителем процесса с повышенными привилегиями, AIS использует новую поддержку в API-функции CreateProcessAsUser, которая устанавливает идентификатор для родительского процесса на тот процесс, который изначально запускал эту функцию. (Дополнительные сведения о процессах и об этом механизме даны в главе 5 «Процессы и потоки».) Именно поэтому в таких утилитах, как Process Explorer, показывающих древовидную структуру процессов, процесс с повышенными привилегиями не показывается в качестве дочернего для хост-процесса службы AIS. На рис. 6.22 показаны операции, задействованные в запуске процесса с повышенными привилегиями под учетной записью обычного пользователя.

Запрос административных прав

Существует несколько способов идентификации потребности системы и приложений в административных правах. Один из них, который проявляется в пользовательском интерфейсе Explorer, является командой контекстного меню Запуск от имени администратора (Run As Administrator) и ее вариантом использования в ярлыке. Эти элементы также имеют в своем составе значок сине-золотистого щита, который должен помещаться вместе с любой кнопкой или пунктом меню, приводящему при их задействовании к повышению привилегий. Выбор команды Запуск от имени администратора (Run As Administrator) заставляет Explorer вызвать API-функцию ShellExecute с глаголом «runas».

Основная масса программ установки требует административных прав, то же самое относится и к загрузчику образов, который инициирует запуск исполняющей системы, включает код обнаружения установщика для определения вероятности наличия устаревших установщиков. Некоторые из используемых им

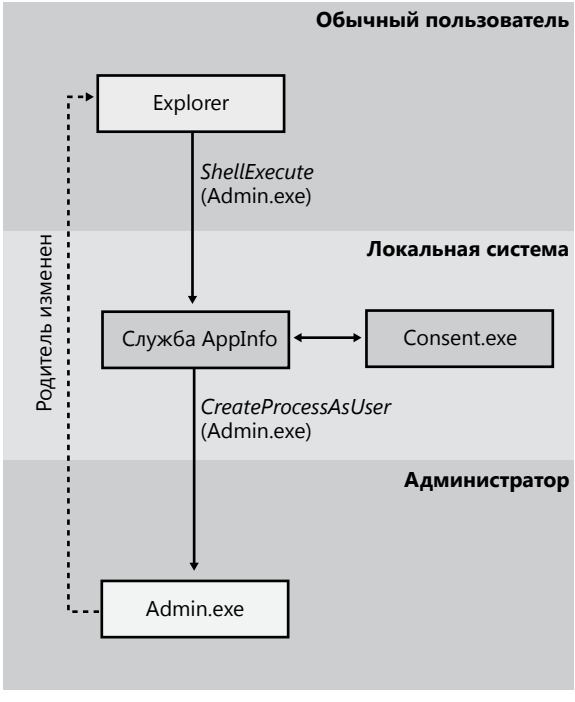


Рис. 6.22. Запуск административного приложения от имени обычного пользователя

эвристических правил выполняют сравнительно несложные задачи обнаружения внутренней информации о версии или обнаружения в имени файла образа таких слов, как setup (настройка), install (установка) или update (обновление). Более сложные средства обнаружения включают сканирование байтовых последовательностей в исполняемой части кода, которые являются общими последовательностями для оболочек установки, разработанных сторонними производителями. Загрузчик образа также вызывает библиотеку совместимости приложений, чтобы посмотреть, не требует ли целевой исполняемый файл прав администратора. Библиотека просматривает базы данных совместимости приложений, чтобы установить, есть ли у исполняемого образа связанные с ним флаги совместимости RequireAdministrator или RunAsInvoker.

Наиболее распространенным способом востребования исполняемым образом прав администратора является включение тега requestedExecutionLevel в файл манифеста приложения. Принадлежащий элементу атрибут уровня может иметь одно из трех значений, показанных в табл. 6.12.

Присутствие в манифесте элемента trustInfo (который можно увидеть в извлечении из рассматриваемого далее дампа eventvwr.exe) свидетельствует о том, что исполняемый образ был написан с поддержкой UAC и в него вложен элемент requestedExecutionLevel. Атрибут uiAccess находится там, где доступные приложения могут воспользоваться обходом упомянутой ранее изоляции привилегий пользовательского интерфейса (UIPI).

Таблица 6.12. Запрашиваемые уровни повышения привилегий

Уровень повышения	Значение	Использование
As Invoker (с правами процесса-родителя)	Права администратора не нужны; просьба на повышение никогда не высказывается	Обычные пользовательские приложения, которым не требуются административные привилегии, например Блокнот
Highest (наивысший)	Доступно согласие на допустимый запрос — Available Request для наивысших прав. Если пользователь вошел в систему как обычный пользователь, процесс будет запущен с правами процесса-родителя; в противном случае появится AAM-приглашение на повышение привилегий, и процесс будет запущен с полными административными правами	Приложения, которые могут работать без полных административных прав, но предполагают, что пользователям нужен полный доступ, если он дается без особого труда. Например, этот уровень используют Редактор реестра, Консоль управления Microsoft Management Console и просмотрщик событий Event Viewer
Require Administrator (с требованием прав администратора)	Всегда запрашивает административные права — для обычных пользователей будет показано диалоговое окно приглашения OTS, в противном случае будет показано диалоговое окно AAM	Приложения, требующие для работы административных прав, например Редактор настроек брандмауэра, который влияет на безопасность всей системы

```
C:\>strings c:\Windows\System32\eventvwr.exe
...
<trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
    <security>
        <requestedPrivileges>
            <requestedExecutionLevel
                level="highestAvailable"
                uiAccess="false"
            />
        </requestedPrivileges>
    </security>
</trustInfo>
<asmv3:application>
   <asmv3:windowsSettings xmlns="http://schemas.microsoft.com/SMI/2005/
WindowsSettings">
        <autoElevate>true</autoElevate>
   </asmv3:windowsSettings>
</asmv3:application>
...
```

Самым простым способом обнаружения значений, указанных исполняемым образом, является просмотр его манифеста с помощью утилиты Sysinternals Sigcheck:

```
sigcheck –m <executable>
```

ЭКСПЕРИМЕНТ: ИСПОЛЬЗОВАНИЕ ФЛАГОВ СОВМЕСТИМОСТИ ПРИЛОЖЕНИЙ

В данном эксперименте мы воспользуемся флагами совместимости приложений для запуска Редактора реестра под учетной записью обычного пользователя. Это позволит обойти флаг RequireAdministrator и заставит включить виртуализацию для Regedit.exe, позволяя вам внести изменения непосредственно в виртуализированный реестр.

1. Перейдите в каталог %SystemRoot% и скопируйте файл Regedit.exe в другой путь на вашей системе (например, в C:\ или в папку вашего рабочего стола).
2. Перейдите к разделу реестра HKLM\Software\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Layers и создайте новый строковый параметр с именем пути, в который вы скопировали Regedit.exe, например c:\regedit.exe
3. Установите для параметра этого раздела значение RUNASINVOKER.

Теперь запустите Regedit.exe из того места, где он находится. (Сначала обязательно закройте любые запущенные копии Редактора реестра.) Теперь вы увидите обычное диалоговое окно AAM, и Regedit.exe будет работать с правами обычного пользователя. Теперь у вас также появится возможность посмотреть на виртуализированный реестр, то есть теперь можно будет посмотреть, что видят устаревшие приложения при доступе к реестру. ■

Автоповышение привилегий

В исходной конфигурации (сведения о ее изменении будут даны в следующем разделе), многие исполняемые файлы Windows и аплеты панели управления не выдают приглашение на повышение привилегий для пользователей-администраторов, даже если им для работы нужны административные права. Так происходит благодаря механизму под названием автоповышение привилегий, который предназначен для исключения пользователей-администраторов из числа тех, кому показываются приглашения на повышение привилегий для большинства случаев их работы; программы будут автоматически запускаться под полным административным маркером пользователя.

У автоповышения есть несколько требований. Интересующий исполняемый файл должен рассматриваться в качестве исполняемого файла Windows. Это означает, что он должен быть подписан издателем Windows (не только Microsoft) и он должен находиться в одном из нескольких считающихся безопасными каталогов: %SystemRoot%\System32 и в большинстве его подкаталогов, %Systemroot%\Ehome и в небольшом количестве каталогов, находящихся в каталоге %ProgramFiles%, например в тех, в которых содержится защитник Windows — Windows Defender и журнал Windows — Windows Journal.

Есть еще и дополнительные требования, которые зависят от типа исполняемого файла.

Файлы с расширением .exe, за исключением Mmc.exe, получают автоповышение привилегий, если таковое ими запрашивается через элемент autoElevate в своих манифестах. Это проиллюстрировано в предыдущем разделе в строковом дампе файла EventVwr.exe.

Windows также включает краткий внутренний список исполняемых файлов, получающих автоповышение привилегий без элемента autoElevate. Двумя примерами могут послужить Spinstall.exe, установщик пакетов обновлений, и Pkgmgr.exe, диспетчер пакетов. Они обрабатываются таким образом, потому что они также поставляются в качестве внешних компонентов Windows 7; они должны иметь возможность работать на более ранних версиях Windows, где элемент autoExecute в их манифестах может привести к ошибке. Эти исполняемые файлы должны, кроме того, соответствовать ранее рассмотренным требованиями к исполняемым файлам Windows, касающимся цифровой подписи и каталога.

Файл Mmc.exe рассматривается в качестве особого случая, поскольку решение вопроса об автоповышении для него привилегий зависит от того, какую оснастку управления системой он должен загрузить. Обычно Mmc.exe вызывается с командной строкой, указанной в файле с расширением .msc, которая, в свою очередь, указывает на загружаемую оснастку. Когда консоль Mmc.exe запускается из защищенной учетной записи администратора (работающей с ограниченным маркером администратора), она запрашивает у Windows административные права. Windows проверяет тот факт, что Mmc.exe является исполняемым файлом Windows, а затем проверяет файл с расширением .msc. Этот файл также должен пройти тесты для исполняемых файлов Windows executable, и, более того, он должен быть внесен во внутренний список файлов с расширением .msc, для которых применяется автоповышение привилегий.

И наконец, административные права могут запрашиваться COM-объектами в разделах реестра. Для этого требуется подраздел Elevation с REG_DWORD-параметром под названием Enabled, имеющим значение 1. Как COM-объект, так и создающий его экземпляр исполняемый файл должны отвечать требованиям, предъявляемым Windows к исполняемым файлам, хотя исполняемый файл не нуждался в запросе автоповышения привилегий.

Управление поведением UAC

Изменить поведение UAC можно с помощью диалогового окна, показанного на рис. 6.23. Это диалоговое окно можно вызвать, воспользовавшись пунктами меню Панель управления (Control Panel) ▸ Центр поддержки (Action Center) ▸ Изменение параметров контроля учетных записей (Change User Account Control Settings). Управление в своей исходной позиции для Windows 7 показано на рис. 6.23.

Здесь работают четыре возможных параметра, описание которых дано в табл. 6.13.

Использовать третью позицию не рекомендуется, потому что UAC-приглашение на повышение привилегий появляется не на защищенном рабочем столе, а на обычном рабочем столе пользователя. Это может позволить вредоносным программам, запущенным в том же сеансе, изменить появление приглашения. Эта настройка предназначена для использования только на тех системах, где видеоподсистемой затрачивается много времени на затенение рабочего стола или испытываются какие-то другие неудобства для обычного отображения, выполняемого UAC.

Использовать самую низкую позицию крайне нежелательно, поскольку при ее установке система UAC с точки зрения административных учетных записей полностью отключается. Все процессы, запускаемые пользователем под учетной

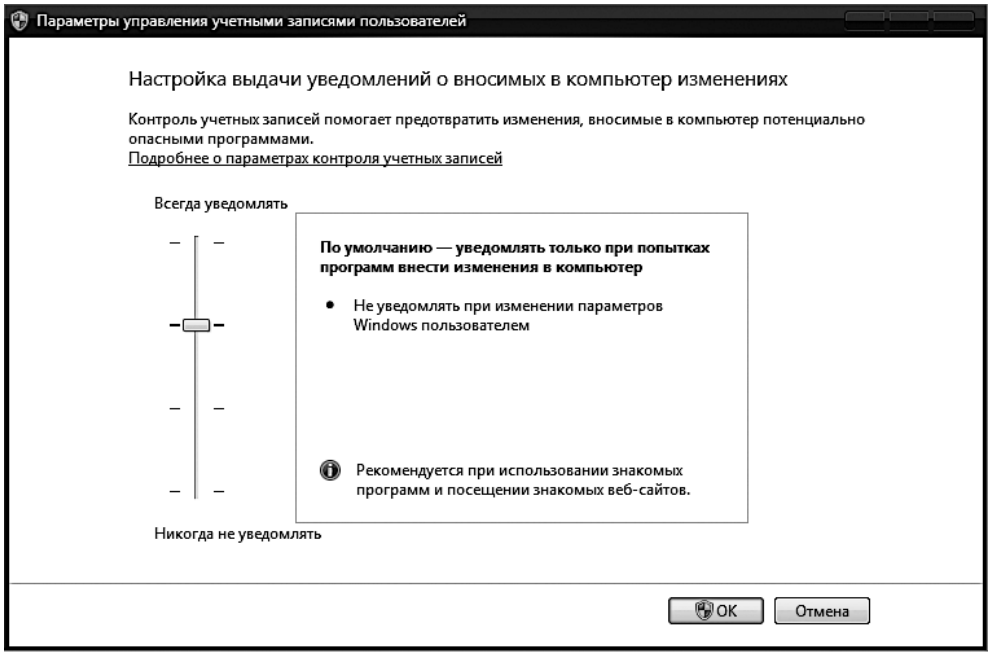


Рис. 6.23. Настройки управления учетными записями пользователей

Таблица 6.13. Параметры управления учетными записями пользователей

Позиция ползунка	**Когда пользователь-администратор не работает с административными правами…**		**Примечания**
	…попытки изменить настройки Windows, например, используя конкретные аплеты Панели управления приводят к тому, что	**… попытки установки программного обеспечения или запуска программы, чей манифест требует повышения привилегий, или с использованием команды Запустить от имени Администратора (Run As Administrator) приводят к тому, что**	
Самая высокая позиция («Всегда уведомлять»)	UAC-приглашение на повышение привилегий появляется на защищенном рабочем столе	UAC-приглашение на повышение привилегий появляется на защищенном рабочем столе	Была характерна для Windows Vista
Вторая позиция	Повышение привилегий UAC происходит автоматически без приглашения или уведомления	UAC-приглашение на повышение привилегий появляется на защищенном рабочем столе	Исходная установка Windows 7

Третья позиция	Повышение привилегий UAC происходит автоматически без приглашения или уведомления	UAC-приглашение на повышение привилегий появляется на обычном рабочем столе пользователя	Не рекомендуется
Самая низкая позиция («Никогда не уведомлять»)	UAC для пользователей-администраторов выключен	UAC для пользователей-администраторов выключен	Не рекомендуется

Таблица 6.14. Параметры реестра, связанные с управлением учетными записями пользователей

Позиция ползунка	ConsentPromptBehaviorAdmin	ConsentPromptBehaviorUser	EnableLUA	PromptOnSecureDesktop
Самая высокая позиция («Всегда уведомлять»)	2 (показывать ААС UAC-приглашение на повышение привилегий)	3 (показывать OTS UAC-приглашение на повышение привилегий)	1 (включено)	1 (включено)
Вторая позиция	5 (показывать ААС UAC-приглашение на повышение привилегий, за исключением попыток изменения настроек Windows)	3	1	1
Третья позиция	5	3	1	0 (отключено; UAC-приглашение появляется на обычном рабочем столе пользователя)
Самая низкая позиция («Никогда не уведомлять»)	0	3	0 (отключено. Вход в систему под административными учетными записями не приводит к созданию ограниченных административных маркеров)	0

записью администратора, будут выполняться с полными правами пользователя-администратора, без фильтрованных административных маркеров. Для этих учетных записей также отключена виртуализация реестра и файловой системы и отключен защищенный режим Internet Explorer. Но виртуализация по-прежнему действует для неадминистративных учетных записей, и для таких учетных записей будут по-прежнему выводиться OTS-приглашения на повышение привилегий в случае, если будут предприняты попытки изменения настроек Windows, запуска программы, требующей повышения привилегий или использования в Explorer пункта контекстного меню Запуск от имени Администратора (Run As Administrator).

Как показано в табл. 6.14, настройки UAC хранятся в разделе HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System в четырех параметрах реестра. Параметр ConsentPromptBehaviorAdmin управляет UAC-приглашением на повышение привилегий для администраторов, работающих с фильтрованным административным маркером, а параметр ConsentPromptBehaviorUser управляет UAC-приглашением для пользователей, не являющихся администраторами.

Идентификация приложений (AppID)

Исторически решения в сфере безопасности в Windows основывались на идентификации пользователя (в форме пользовательского SID и принадлежности к группе), но растущее количество компонентов системы безопасности (AppLocker, брандмауэр, антивирус, противодействие вредоносным программам, службы управления правами — Rights Management Services и т. д.) вынуждает принимать решения по безопасности на основе выполняемого кода. В прошлом каждый из этих компонентов безопасности использовал свой собственный особый метод для идентификации приложений, что приводит к несовместимости и слишком сложной политики авторства. Цель AppID заключается во внесении согласованности в способах распознавания приложений компонентами системы безопасности путем предоставления единого набора API-интерфейсов и структур данных.

ПРИМЕЧАНИЕ

Это не то же самое, что система AppID, используемая приложениями DCOM/COM+, где GUID представляет процесс, совместно используемый несколькими CLSID-идентификаторами, и это не AppID, используемая приложениями Windows Live.

Так же как пользователь идентифицируется при входе в систему, приложение идентифицируется перед запуском путем генерирования AppID основной программы. Идентификатор AppID может быть сгенерирован из любого из следующих атрибутов приложения:

- Поля внутри сертификата подписи кода, встроенного в файл, позволяют получить различные комбинации имени издателя, имени продукта, имени файла и версии. APPID://FQBN является полностью определенным двоичным именем — Fully Qualified Binary Name, представленным в виде строки следующего формата: {*Издатель**Продукт**Имя_файла*,*Версия*}. Имя издателя является полем Subject сертификата x.509, использованного для подписи кода, при-

меняющего следующие поля: O = Organization (организация), L = Locality (локализация), S = State (штат или провинция) и C = Country (страна).

- Хэши файлов. Существует ряд методов, которые можно использовать для создания хэшей. Исходным методом является APPID://SHA256HASH. Но для обратной совместимости с SRP и большинством сертификатов x.509 по-прежнему поддерживается SHA-1 (APPID://SHA1HASH). APPID://SHA256HASH указывает на SHA-256 хэш файла.

Частичный или полный путь к файлу. APPID://Path указывает путь с дополнительными символами-заместителями («*»).

ПРИМЕЧАНИЕ

AppID не служит средством сертификации качества или безопасности приложения, это просто способ идентификации, чтобы администраторы могли ссылаться на приложение в решениях политики безопасности.

AppID хранится в маркере доступа процесса, позволяя любым компонентам системы безопасности принять решения по авторизации на основе единой согласованной идентификации. Механизм AppLocker использует условные ACE-элементы (рассмотренные ранее) для определения, разрешено ли конкретной программе быть запущенной тем или иным пользователем.

Когда AppID создается для файла, имеющего цифровую подпись, сертификат этого файла кэшируется и сличается с доверенным корневым сертификатом. Путь сертификата ежедневно перепроверяется, чтобы убедиться, что путь сертификата остался правильным. Кэширование и проверка сертификата записываются в системный журнал событий (рис. 6.24).

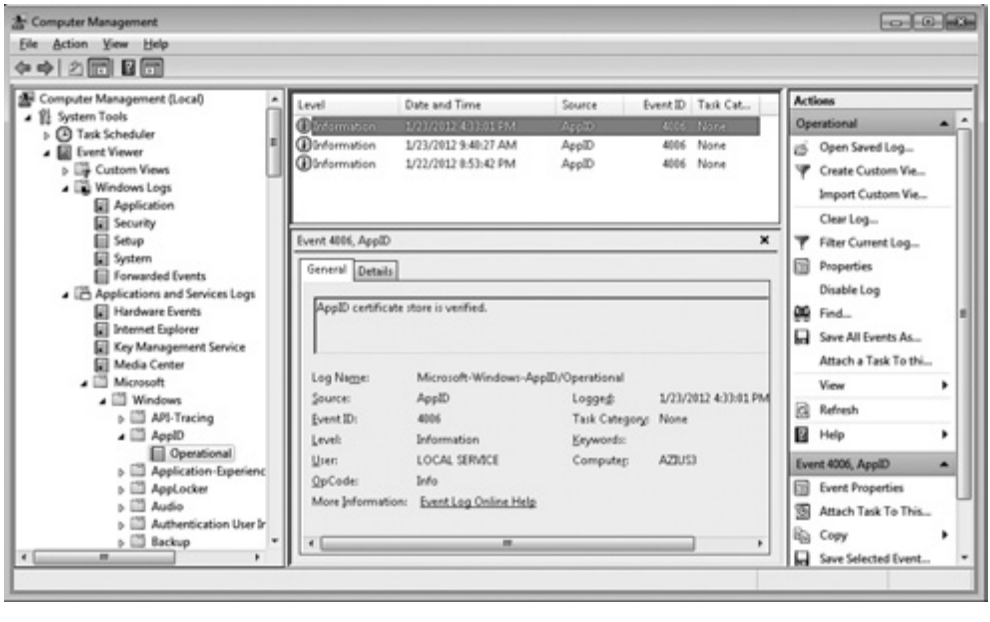


Рис. 6.24. Просмотрщик событий, показывающий службу AppID, которая проверяет подпись программы.

AppLocker

Новым в Windows 7 и в Windows Server 2008/R2 (в выпусках Enterprise и Ultimate) стало свойство, известное как AppLocker, позволяющее администратору заблокировать систему для предотвращения запуска неавторизованных программ. В Windows XP появились политика ограниченного использования программ — Software Restriction Policies (SRP), — которая была первым шагом по направлению к этой возможности, но SRP испытывала трудности управления, и она не могла применяться к конкретным пользователям или группам. От правил SRP страдали все пользователи. AppLocker пришел на замену SRP, и пока сосуществует рядом с SRP, а правила AppLocker хранятся отдельно от правил SRP. Если и правила AppLocker, и правила SRP находятся в одном и том же объекте групповой политики — Group Policy object (GPO), то будут применяться только правила AppLocker. Другими свойствами, дающими превосходство AppLocker над SRP, является режим аудита AppLocker, позволяющий администратору создавать политику AppLocker и проверять результаты (хранящиеся в системном журнале событий) для определения, будет ли политика выполняться, как ожидалось, фактически не накладывая ограничений. Режим аудита AppLocker может использоваться для отслеживания того, какие приложения используются одним или несколькими пользователями системы.

AppLocker позволяет администратору ограничить запуск файлов следующих типов:

- исполняемых образов (.EXE и .COM);
- динамически подключаемых библиотек (.DLL и .OCX);
- установщика Microsoft Software Installer (.MSI и .MSP) как для установки, так и для удаления установки;
- сценарии Windows PowerShell (.PS1);
- пакетные (.BAT и .CMD);
- сценарии VisualBasic (.VBS);
- сценарии JavaScript (.JS).

AppLocker предоставляет простой GUI-механизм на основе правил, которые очень похожи на правила сетевого брандмауэра, для определения, каким приложениям или сценариям разрешено запускаться конкретными пользователями и группами, используя условные ACE-элементы и AppID-атрибуты. В AppLocker используются правила двух типов:

- разрешающие конкретным файлам запускаться, не признавая больше ничего;
- запрещающие конкретным файлам запускаться, разрешая все остальное. «Запрещающие» правила имеют приоритет над «разрешающими».

У каждого правила может также быть список исключений для удаления файлов из правила. Используя исключение, можно создать правило «Разрешить запускать все в каталогах C:\Windows или C:\Program Files, за исключением встроенных игр».

Правила AppLocker могут быть связаны с конкретным пользователем или группой. Это позволяет администратору поддерживать требования совместимости путем проверки и принудительной установки круга пользователей, имеющих

возможность запуска определенных приложений. Например, можно создать правило «Разрешить пользователям, принадлежащим к группе Finance security, запускать приложения финансового направления». Это правило заблокирует всем, кто не входит в группу Finance security, возможность запуска финансовых приложений (включая администраторов), но по-прежнему предоставит доступ тем, у кого есть деловые потребности в запуске приложений. Другим полезным правилом станет предохранение пользователей из группы Receptionists от установки или запуска не получивших одобрения приложений.

Правила AppLocker зависят от условных ACE-элементов и атрибутов, определяемых AppID. Правила могут быть созданы с использованием следующих критериев:

- Полей внутри сертификата, подписывающего код и встроенного в файл, позволяющих составлять различные комбинации из имени издателя, имени продукта, имени файла и версии. Например, правило может быть создано для того, чтобы «Разрешить запускаться всем версиям Contoso Reader, чей номер выше 9.0» или «Разрешить всем из группы graphics запускать установщик или приложение от Contoso под названием GraphicsShop, если версия имеет номер 14.*». Например, следующая SDDL-строка запрещает пользователю, вошедшему в систему под учетной записью RestrictedUser (идентифицируемой по SID пользователя), доступ для выполнения к любым, имеющим цифровую подпись программам, изданным Contoso:

```
D:(XD;;FX;;;S-1-5-21-3392373855-1129761602-2459801163-1028;
((Exists APPID://FQBN)
&& ((APPID://FQBN) >= ({"O=CONTOSO, INCORPORATED, L=REDMOND,
    S=CWASHINGTON, C=US\*\*",0})))
```

- Пути к каталогу, разрешающему запускаться только тем файлам, которые находятся внутри определенного дерева каталогов. Этот критерий может также использоваться для идентификации конкретных файлов. Например, следующая SDDL-строка запрещает пользователю, вошедшему в систему под учетной записью RestrictedUser (идентифицируемой по SID пользователя), доступ для выполнения к программам в каталоге C:\Tools:

```
D:(XD;;FX;;;S-1-5-21-3392373855-1129761602-2459801163-1028;(APPID://PATH
Contains "%OSDRIVE%\TOOLS\*"))
```

- Хэша файла. Использование хэша также позволит обнаружить внесение в файл изменений и помешать его запуску, что также может рассматриваться как недостаток, если файлы часто подвергаются изменениям, поскольку правило на основе хэша нужно будет часто обновлять. Хэши файлов часто используются для сценариев, поскольку цифровую подпись имеют лишь некоторые сценарии. Например, следующая SDDL-строка запрещает пользователю, вошедшему в систему под учетной записью RestrictedUser (идентифицируемой по SID пользователя), доступ для выполнения программ с указанными значениями хэша:

```
D:(XD;;FX;;;S-1-5-21-3392373855-1129761602-2459801163-1028;(APPID://SHA256HASH
Any_of {#7a334d2b99d48448eedd308dfca63b8a3b7b44044496ee2f8e236f5997f1b647,
#2a782f76cb94ece307dc52c338f02edbbfdca83906674e35c682724a8a92a76b}))
```

На локальной машине правила AppLocker могут быть определены с помощью MMC-оснастки Локальная политика безопасности (Security Policy, %SystemRoot%\System32\secpol.msc) или с помощью сценария Windows PowerShell или может быть навязана машинам, принадлежащим домену с использованием групповой политики. Правила AppLocker хранятся в нескольких местах реестра:

- HKLM\Software\Policies\Microsoft\Windows\SrpV2. Этот раздел также зеркально копируется в разделе HKLM\SOFTWARE\Wow6432Node\Policies\Microsoft\Windows\SrpV2. Правила хранятся в формате XML.
- HKLM\SYSTEM\CurrentControlSet\Control\Srp\Gp\Exe. Правила хранятся в виде SDDL и двоичного ACE.
- HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Group PolicyObjects\{GUID}Machine\Software\Policies\Microsoft\Windows\SrpV2. AppLocker-политика, навязываемая доменом как часть объекта групповой политики — Group Policy Object (GPO), хранится здесь в формате XML.

Сертификаты для файлов, которые были запущены, кэшируются в реестре в разделе HKLM\SYSTEM\CurrentControlSet\Control\AppID\CertStore. AppLocker также создает цепочку сертификатов (хранящуюся в разделе HKLM\SYSTEM\

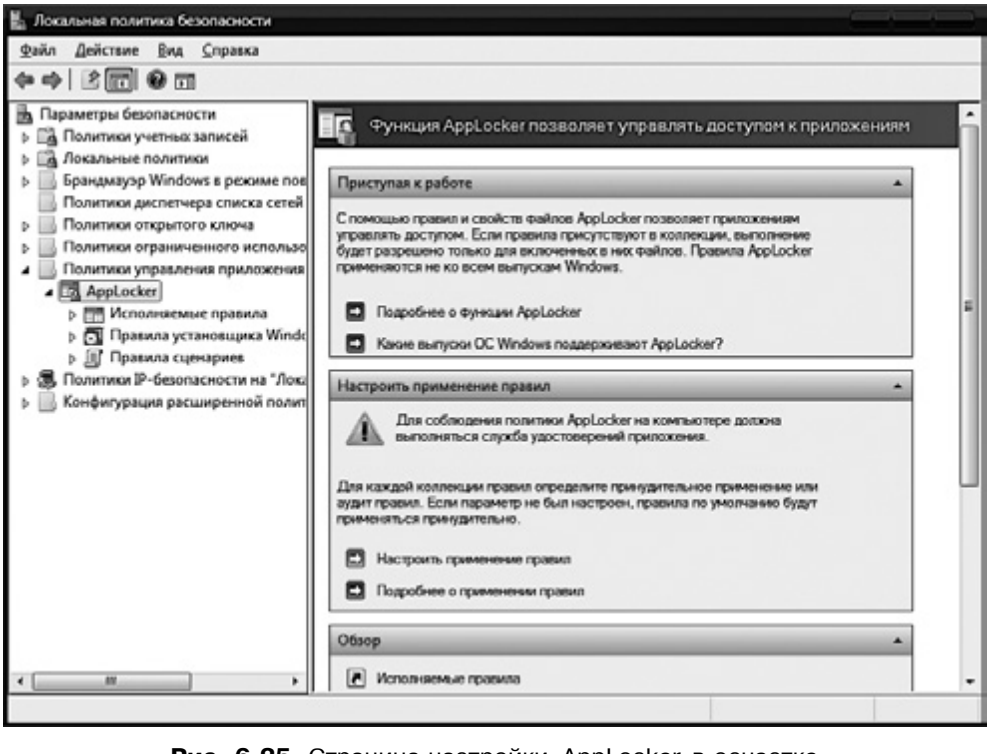


Рис. 6.25. Страница настройки AppLocker в оснастке Локальная политика безопасности

CurrentControlSet\Control\AppID\CertChainStore) от сертификата, найденного в файле, назад к доверенному корневому сертификату (рис. 6.25).

Есть также ориентированные на AppLocker команды PowerShell (известные также как командлеты — cmdlets), позволяющие вести развертывание и тестирование с помощью сценариев. На рис. 6.26 демонстрируется использование команд PowerShell для определения файлов в дереве каталогов, имеющих цифровые подписи, сохранения текущей политики AppLocker в XML-файле и показа, какие исполняемые файлы в дереве каталогов могут быть запущены пользователем RestrictedUser.

Службы AppID и SRP сосуществуют в одном и том же двоичном файле (%SystemRoot%\System32\AppIdSvc.dll), который запускается в процессе SvcHost. Служба запрашивает уведомление о внесении изменений в реестр для отслеживания любого изменения в тех разделах, которые записаны либо GPO, либо AppLocker UI в MMC-оснастке Локальная политика безопасности. Когда обнаруживается изменение, служба AppID запускает задание пользовательского режима (%SystemRoot%\System32\AppIdPolicyConverter.exe), которое читает новые XML-правила и переводит их в двоичный формат ACE-элементов и SDDL-строки, доступные для понимания компонентами AppID и AppLocker, относящимися как к пользовательскому режиму, так и к режиму ядра. Задание сохраняет переведенные правила в разделе HKLM\SYSTEM\CurrentControlSet\Control\Srp\Gp. Этот раздел доступен по записи только для групп Система (SYSTEM) и Администраторы (Administrators) и помечен для аутентифицированных пользо-

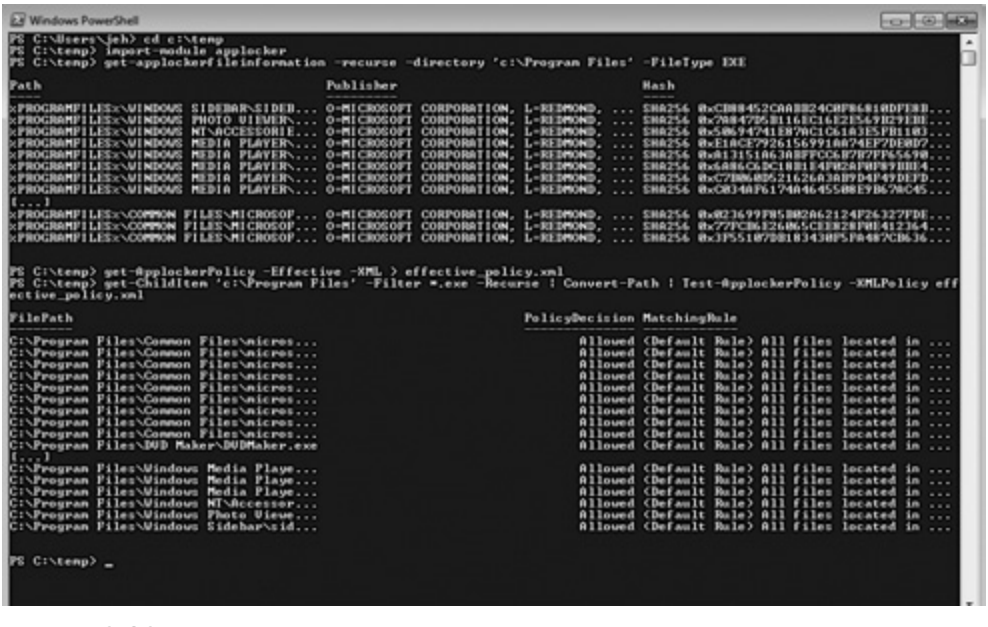


Рис. 6.26. Командлет Powershell, используемый для проверки исполняемых файлов на наличие цифровых подписей, сохранения политики AppLocker в XML-файле, и тестирования возможности конкретного пользователя запускать исполняемые файлы

вателей как раздел только для чтения. Компоненты AppID как пользовательского режима, так и режима ядра считывают переведенные правила непосредственно из реестра. Служба также отслеживает хранилище доверенных корневых сертификатов локальной машины и вызывает задание пользовательского режима (%SystemRoot%\System32\AppIdCertStoreCheck.exe) для повторной проверки сертификатов как минимум ежедневно, а также при каждом изменении в хранилище сертификатов. Драйвер AppID режима ядра (%SystemRoot%\System32\drivers\AppId.sys) уведомляется об изменении правил службой AppID посредством запроса APPID_POLICY_CHANGED DeviceIoControl (рис. 6.27).

Администратор может отследить, какие приложения были разрешены или запрещены, просмотрев системный журнал событий (рис. 6.28), используя просмотрщик событий (если AppLocker был настроен и служба запущена).

Реализации AppID, AppLocker и SRP не имеют четко обозначенных границ и нарушают принципы строгого разделения по уровням, здесь различные логические компоненты сосуществуют в одних и тех же исполняемых файлах, и система присваивания имен не столь последовательна, как хотелось бы.

Служба AppID запускается как LocalService, поэтому она имеет доступ к имеющемуся в системе хранилищу доверенных корневых сертификатов — Trusted Root Certificate Store. Это также позволяет ей выполнять проверку сертификатов. Служба AppID отвечает за следующие действия:

- проверку сертификатов издателей;
- добавление в кэш новых сертификатов;
- обнаружение обновления правила AppLocker и уведомление драйвера AppID.

Драйвер AppID выполняет основную часть действий AppLocker и зависит от обмена данными (через запросы DeviceIoControl) со службой AppID, поэтому его объект устройства защищен с помощью ACL, предоставляя доступ только группам NT SERVICE\AppIDSvc, NT SERVICE\LOCAL SERVICE и BUILTIN\Administrators. Таким образом, драйвер не может быть обманут вредоносным кодом.

При первой загрузке драйвера AppID он запрашивает создание процесса функции обратного вызова (CreateProcessNotifyEx) путем вызова процедуры PsSetCreateProcessNotifyRoutineEx. Когда вызывается процедура CreateProcessNotifyEx, ей передается структура PPS_CREATE_NOTIFY_INFO (описывающая создаваемый процесс). Затем она собирает атрибуты AppID, идентифицирующие исполняемый образ, и записывает их в маркер доступа процесса. Затем она вызывает недокументированную процедуру SeSrpAccessCheck, которая проверяет маркер процесса и условные ACE-элементы правил AppLocker, и определяет, нужно ли разрешать запуск процесса. Если запуск процесса не должен быть разрешен, драйвер записывает STATUS_ACCESS_DISABLED_BY_POLICY_OTHER в поле статуса (Status) структуры PPS_CREATE_NOTIFY_INFO, что приводит к прекращению создания процесса (и устанавливает окончательный статус завершения процесса).

Для накладывания ограничений на выполнение DLL при загрузке DLL в процесс загрузчик образов отправит запрос DeviceIoControl драйверу AppID. Затем драйвер проверит идентичность DLL, проведя сравнение с условными ACE-элементами AppLocker, точно так же, как он бы это сделал в отношении исполняемого файла.

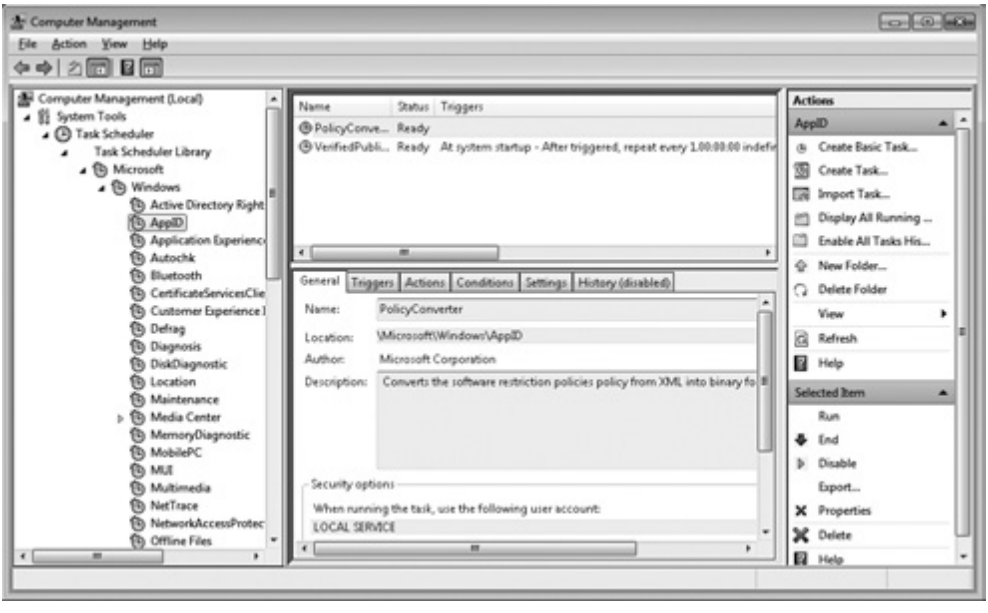


Рис. 6.27. Запланированное задание, запускающееся каждый день для преобразования политик ограничения программного обеспечения, сохраненных в формате XML в двоичный формат

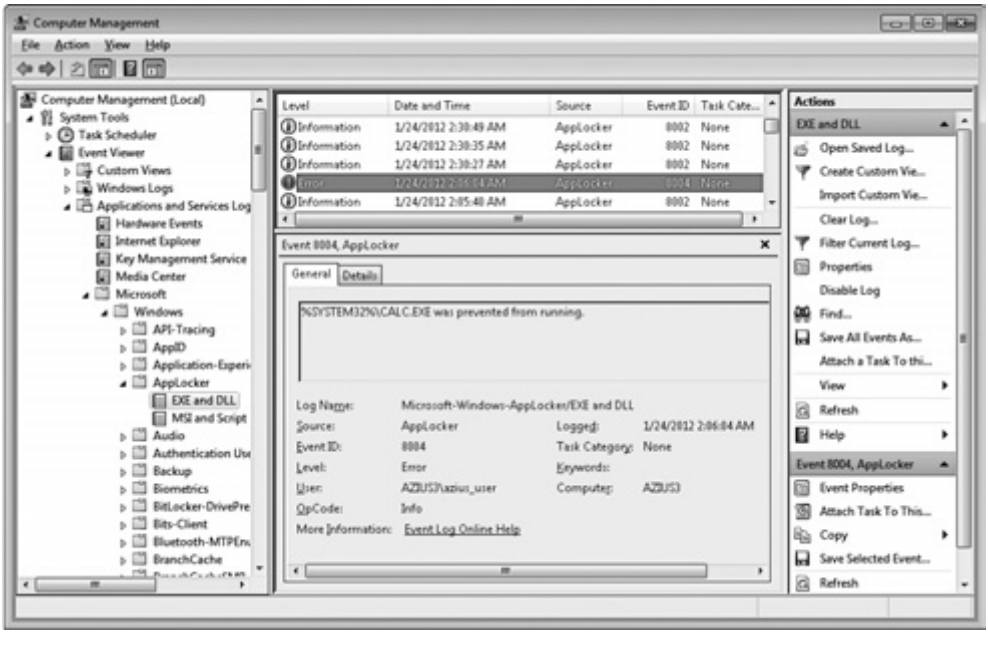


Рис. 6.28. Просмотрщик событий, показывающий AppLocker, разрешающий и запрещающий доступ к различным приложениям. Идентификатор события Event ID со значением 8004 означает «запрещен», а со значением 8002 означает «разрешен»

ПРИМЕЧАНИЕ

Выполнение этих проверок для каждой загрузки DLL требует времени, что может не пройти незамеченным для конечных пользователей. Поэтому правила, касающиеся DLL, обычно отключены, и они могут быть специально включены через вкладку Дополнительно (Advanced) страницы свойств AppLocker в оснастке Локальная политика безопасности (Local Security Policy).

Машина сценариев и MSI-установщик были изменены для вызова API-функций SRP пользовательского режима, как только они открывают файл, чтобы проверить, разрешено ли открытие файла. API-функции SRP пользовательского режима вызывают API-функции AuthZ для выполнения проверки прав доступа с помощью условного ACE-элемента.

Политики ограниченного использования программ

Windows также содержит механизм пользовательского режима, который называется политиками ограниченного использования программ — Software Restriction Policies — и позволяет администраторам управлять тем, какие образы и сценарии выполняются на их системах. Политики ограниченного использования программ являются узлом редактора локальной политики безопасности, показанным на рис. 6.29, которые служат в качестве интерфейса управления политиками исполняемого кода для всей машины, хотя для каждого пользователя политики также доступны с использованием доменных групповых политик.

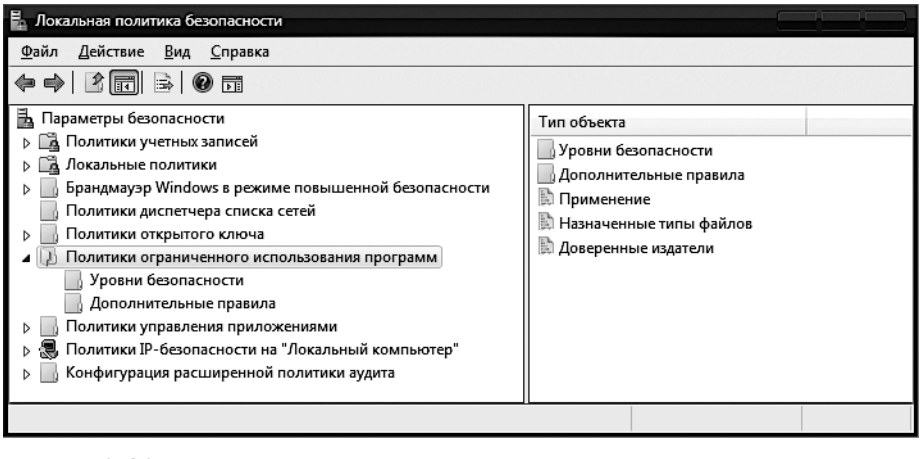


Рис. 6.29. Настройка политик ограниченного использования программ

Ниже узла политик ограниченного использования программ появляется ряд настроек глобальной политики:

- Политика Применение (Enforcement) настраивает применение политик к библиотекам, например к DLL, и определяет применение политик только к пользователям или к администраторам в том числе.

- Политика Назначенные типы файлов (Designated File Types) записывает расширения файлов, которые считаются исполняемым кодом.
- Политика Доверенные издатели (Trusted Publishers) управляет тем, кто может отмечать, какие из сертификатов издателей являются доверенными.

При настройке политики для конкретного сценария или образа администратор может направить систему на его распознавания с использованием его пути, его хэша, его зоны Интернета (в соответствии с ее определениями в Internet Explorer) или его криптографического сертификата, и он может определить, связан ли сценарий или образ с политиками безопасности Не разрешено (Disallowed) или Неограниченный (Unrestricted).

Политика Применение (Enforcement), входящая в политики ограниченного использования программ, применяется в отношении различных компонентов, где файлы рассматриваются как содержащие исполняемый код. Вот перечень некоторых из таких компонентов:

- Windows-функция пользовательского режима CreateProcess, которая находится в библиотеке %SystemRoot%\System32\Kernel32.dll, применяет ее в отношении исполняемых образов.
- Код загрузки DLL в библиотеке Ntdll (%SystemRoot%\System32\Ntdll.dll) применяет ее для DLL-библиотек.
- Окно командной строки Windows (%SystemRoot%\System32\Cmd.exe) применяет ее для файлов с расширениями, указывающими на их пакетную природу.
- Компоненты Windows Scripting Host, запускающие сценарии — %SystemRoot%\System32\Cscript.exe (для сценариев командной строки), %SystemRoot%\System32\Wscript.exe (для UI-сценариев) и %SystemRoot%\System32\Scrobj.dll (для объектов сценариев), — применяют ее для файлов с расширениями, указывающими на их принадлежность к сценариям.

Каждый из этих компонентов определяет, включены ли политики ограничений, считывая значение параметра реестра HKEY_LOCAL_MACHINE\Software\Microsoft\Policies\Windows\Safer\CodeIdentifiers\TransparentEnabled, которое при установке в 1 показывает, что политики задействованы. Затем он определяет, соответствует ли код, предназначенный для выполнения одному из правил, указанных в подразделе раздела CodeIdentifiers, и если соответствует, определяет, должно быть разрешено выполнение или нет. Если соответствие не обнаружено, определение разрешения на выполнение осуществляется на основе исходной политики, указанной в значении параметра DefaultLevel раздела CodeIdentifiers.

Политики ограниченного использования программ являются весьма эффективным средством для предотвращения несанкционированного доступа к коду и к сценариям, но только при условии их правильного применения. Если только исходная политика не настроена на запрещение выполнения, пользователь может вносить незначительные изменения в образ, помеченный как запрещенный, то есть он может обойти правило и выполнить этот образ. Например, пользователь может изменить нейтральный байт в образе процесса, чтобы хэш-правило не смогло его распознать, или скопировать файл в другое место, чтобы обойти правило, основанное на пути к файлу.

ЭКСПЕРИМЕНТ: ПРОСМОТР ПРИНУДИТЕЛЬНОГО ПРИМЕНЕНИЯ ПОЛИТИК ОГРАНИЧЕННОГО ИСПОЛЬЗОВАНИЯ ПРОГРАММ

Увидеть принудительное применение политик ограниченного использования программ косвенным образом можно, наблюдая за обращениями к реестру при попытке выполнения образа, который был вами запрещен к выполнению.

1. Запустите оснастку secpol.msc, чтобы открыть редактор локальной политики безопасности, и перейдите к узлу Политики ограниченного использования программ (Software Restriction Policies).
2. Если политики не определены, выберите пункт контекстного меню Создать новые политики (Create New Policies).
3. Создайте основанную на пути к файлу запрещающую политику ограничения для %SystemRoot%\System32\Notepad.exe.
4. Запустите утилиту Process Monitor и установите включающий (include) для фильтр Safer. (Описание утилиты Process Monitor дано в главе 4.)
5. Откройте окно командной строки и запустите Блокнот (Notepad) из приглашения.

Ваша попытка запустить Notepad приведет к выводу сообщения, в котором говорится, что вы не можете выполнить указанную программу, а Process Monitor должен показать окно командной строки (cmd.exe), запрашивающее политики ограничений на локальной машине. ■

Заключение

Windows предоставляет обширный набор функций безопасности, которые отвечают основным требованиям как государственных учреждений, так и коммерческих структур. В данной главе был дан краткий обзор внутренних компонентов, положенных в основу этих функций безопасности. В следующей главе мы рассмотрим сеть.

Глава 7. Сеть

Microsoft Windows была спроектирована с расчетом на работу в сети, и она включает в себя всестороннюю сетевую поддержку, объединенную с системой ввода-вывода и API-функциями Windows. К четырем основным сетевым программным компонентам относятся службы, API-функции, протоколы и драйверы для сетевых адаптеров, и каждый из этих компонентов накладывается поверх следующего с целью создания сетевого стека. Для каждого такого уровня у Windows есть вполне определенные интерфейсы, поэтому в дополнение к использованию разнообразных API-функций, протоколов и драйверов сетевых адаптеров, поставляемых вместе с Windows, сторонние разработчики могут расширить сетевые возможности операционной системы путем разработки своих собственных компонентов.

В данной главе мы пройдем с вами сетевой стек Windows сверху донизу. Сначала будет представлено сопоставление программных сетевых компонентов Windows с исходной моделью взаимодействия открытых систем — Open Systems Interconnection (OSI). Затем последует краткое описание сетевых API-функций, доступных в Windows, с объяснением способов их реализации. Вы узнаете о поддержке множественных перенаправлений и о разрешении имен, увидите, как получить доступ к удаленным файлам и поместить их в кэш, и изучите порядок взаимодействия множества драйверов с целью формирования стека сетевого протокола. После того как будет изучен вопрос реализации драйверов сетевых адаптеров, будет рассмотрено связывание, которое является той самой интегрирующей технологией, которая связывает службы, стеки протоколов и сетевые адаптеры.

Сетевая архитектура Windows

Целью сетевого программного обеспечения является получение запроса (в форме запроса ввода-вывода) от приложения на одной машине, передача его на другую машину, выполнение запроса на удаленной машине и возвращение результата на первую машину. В ходе этого процесса запрос должен быть преобразован несколько раз. Запрос высокого уровня, например «считать *x* байтов из файла *y* на машине *z*», требует наличия программного обеспечения, способного определить, как добраться до машины *z* и какое имеется коммуникационное программное обеспечение, понятное машине. Затем запрос должен быть изменен для передачи по сети, например разбит на короткие информационные пакеты. Когда запрос достигнет противоположной стороны, он должен быть проверен на целостность, декодирован и отправлен надлежащему компоненту операционной системы для выполнения. И наконец, ответ должен быть закодирован для отправки по обратному адресу по сети.

Исходная модель OSI

Чтобы помочь разным производителям компьютерного оборудования привести к единому стандарту и интегрировать их сетевое программное обеспечение, в 1984 году международная организация по стандартам — International

Organization for Standardization (ISO) — определила программную модель для обмена сообщениями между машинами. В результате появилась исходная модель взаимодействия открытых систем — Open Systems Interconnection (OSI). В модели определены шесть уровней программного обеспечения и один физический уровень оборудования, показанные на рис. 7.1.

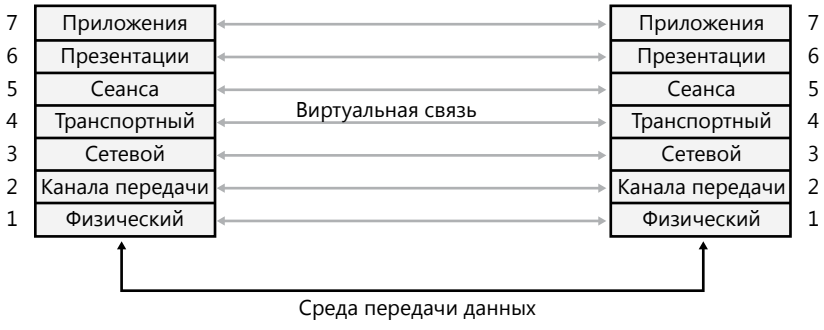


Рис. 7.1. Исходная модель OSI

Исходная модель OSI является идеализированной схемой, которой в точности придерживаются лишь несколько систем, но она зачастую используется в качестве структурной основы для обсуждений принципов построения сетей. Предполагается, что каждый уровень на одной машине «ведет разговор» с таким же уровнем на другой машине. Обе машины на одном и том же уровне «разговаривают» на одном и том же языке, или протоколе. Но в действительности передача по сети должна пройти вниз через каждый уровень на клиентской машине, быть передана по сети, а затем подняться по уровням на целевой машине, пока не будет достигнут уровень, способный понять и выполнить запрос.

Каждый уровень в модели OSI предназначен для предоставления служб для более высоких уровней и для абстрагирования того, как службы реализуются на более низких уровнях. Описание подробностей каждого уровня не входит в сферу вопросов, рассматриваемых в данной книге, но далее будет дано краткое описание каждого уровня, представленного в модели OSI.

ПРИМЕЧАНИЕ

Большинство описаний сетей начинаются с самого верхнего уровня и ведутся сверху вниз до самого нижнего уровня, но здесь описание уровней начнется снизу и будет идти по направлению к верхнему уровню, чтобы показать, как каждый уровень выстраивается на основе служб, предоставляемых уровнями, расположенными ниже его.

- **Физический уровень (Physical).** Это самый нижний уровень в модели OSI, и он проводит обмен сигналами между взаимодействующими сетевыми объектами по какой-нибудь физической среде переноса (по проводам, радиоканалу, оптоволокну или по другому носителю). На физическом уровне определяются механические, электрические, функциональные и процедурные стандарты для доступа к среде переноса, это касается соединителей, кабелей, сигнальных устройств и т. д. Простыми примерами могут послужить Ethernet (IEEE 802.3) и Wi-Fi (IEEE 802.11).

- **Уровень канала передачи данных (Datalink).** На этом уровне происходит обмен блоками данных (которые также называются *пакетами*) между *физически смежными* сетевыми объектами (известными как *станции*) с помощью служб, предоставляемых физическим уровнем. По своей природе уровень канала передачи данных тесно связан с физическим уровнем и, в действительности, является более выраженной архитектурной абстракцией по сравнению с другими уровнями модели. Уровень канала передачи данных предоставляет каждой станции ее собственный уникальный сетевой адрес, а также предоставляет связь между станциями «от точки до точки» (как, к примеру, между двумя системами, подключенными к одной и той же сети Ethernet). Возможности уровня канала передачи данных существенно изменяются в зависимости от физического уровня. Обычно ошибки передачи и приема определяются на уровне канала передачи данных, и в некоторых случаях ошибка может быть исправлена. Уровень канала передачи данных может быть сориентирован на соединения, что обычно используется в глобальных сетях — wide area networks (WAN), или на передачу данных без установки соединения, что обычно используется в локальных сетях — local area networks (LAN). За основные LAN-архитектуры, используемые по всему миру, отвечает комитет 802 Института инженеров по электротехнике и электронике — IEEE (Institute of Electrical and Electronics Engineers), — и именно он определяет физический уровень и уровень канала передачи данных большинства сетевого оборудования. Этот комитет разделил уровень канала передачи данных на два подуровня: управления логической связью — Logical Link Control (LLC), и управления доступом к среде — Medium Access Control (MAC). LLC-уровень предоставляет единый метод доступа к сетевому уровню для обмена данными с любым устройством спецификации 802.x MAC, изолируя сетевой уровень от физического типа LAN. MAC-уровень предоставляет функции управления доступом к общей сетевой среде переноса данных и определяет порядок передачи сигналов, протокол обмена, распознавание адреса, генерацию пакета, CRC-генерацию и т. д. Уровень канала передачи данных не гарантирует доставку пакетов к месту их назначения.
- **Сетевой уровень (Network).** На сетевом уровне реализуются адреса узлов и функции маршрутизации, позволяющие пакетам пройти несколько каналов передачи данных. На этом уровне учитывается топология сети (что скрывает ее от транспортного уровня) и имеется представление о том, как направить пакеты на ближайший маршрутизатор. Любой сетевой объект, имеющий сетевой уровень, уровень канала передачи данных и физический уровень, рассматривается в качестве *узла*, и сетевой уровень может перемещать по сети данные между двумя узлами. На сетевом уровне реализуются два типа узлов: конечные узлы, являющиеся источником или приемником данных, и промежуточные узлы (которые, как правило, называются маршрутизаторами), которые направляют пакеты между конечными узлами. Служба сетевого уровня может быть либо сориентирована на соединение, где все пакеты перемещаются между конечными узлами, следуя по сети по одному и тому же пути, либо сориентирована на передачу данных без установки соединения, где каждый пакет направляется сам по себе. Сетевой уровень не гарантирует доставку пакетов к месту назначения.

- **Транспортный уровень (Transport).** Транспортный уровень предоставляет прозрачный механизм передачи данных между конечными узлами. На отправляющей стороне транспортный уровень получает неструктурированный поток данных от вышестоящего уровня и сегментирует данные в дискретные пакеты, которые могут быть отправлены по сети, используя службы находящегося ниже сетевого уровня. На принимающей стороне транспортный уровень снова собирает пакеты из сетевого уровня в поток данных и предоставляет этот поток вышестоящему уровню. Этот уровень обеспечивает *надежную* передачу данных и будет заново передавать потерянные или искаженные пакеты для обеспечения идентичности полученного потока данных переданному потоку.
- **Уровень сеанса (Session).** Этот уровень реализует *связь* или *канал* между взаимодействующими приложениями. У каждой конечной точки связи есть свой собственный адрес (который часто называют *портом*), уникальный для данной системы. На уровне сеанса предоставляются различные коммуникационные службы, например двунаправленная одновременная передача данных (полнодуплексная), двунаправленная чередующаяся (однодуплексная), или однонаправленная. Как только подключение будет осуществлено, системы обычно начинают периодически отправлять сообщения друг другу, чтобы убедиться в функционировании каждого конечного узла подключения. Если при подключении обнаружится некорректируемая ошибка передачи данных, подключение обычно завершается и происходит отключение.
- **Уровень презентации (Presentation).** Уровень презентации отвечает за защиту информационного содержимого данных, отправляемых по сети. На нем производится форматирование данных, включая такие вопросы, как завершение строки либо парой символов возврата каретки и перевода строки — carriage return и line feed (CR/LF), либо просто символом возврата каретки — carriage return (CR), сжатие или шифрование данных, преобразование двоичных данных из прямого порядка байтов (little-endian) в порядок от старшего к младшему (big-endian) и т. д. Этот уровень не представлен в большинстве стеков сетевых протоколов, поэтому его функции реализованы на уровне приложения.
- **Уровень приложения (Application).** Этот уровень обрабатывает информацию, передаваемую между двумя сетевыми приложениями, включая такие функции, как проверки безопасности, идентификация задействованных машин и инициирование обмена данными. Это протокол, используемый двумя обменивающимися данными приложениями и определяемый приложением.

Серыми линиями на рис. 7.1 показаны протоколы, используемые в передаче запроса к удаленной машине. Как уже говорилось, предполагается, что каждый уровень иерархии общается с аналогичным уровнем на другой машине и использует общий протокол. Коллекция тех протоколов, через которые проходит запрос на своем пути вниз и снова вверх по сетевым уровням, называется *стеком протоколов*.

Не все наборы сетевых протоколов реализуют все уровни модели OSI. (Уровень презентации предоставляется довольно редко.) В частности, стек протокола TCP/IP (который предшествовал модели OSI) весьма слабо соответствует абстракциям OSI. По мере прохождения данных вниз по сетевому стеку каждый уровень добавляет заголовок (и, возможно, концевик) к полезным данным, вы-

страивая структуру, очень похожую на луковые чешуйки. Когда эта структура будет получена на удаленном узле, она пройдет вверх по сетевому стеку, и каждый уровень снимает свой заголовок (и концевик) до тех пор, пока полезные данные не будут доставлены получающему приложению.

Сетевые компоненты Windows

На рис. 7.2 представлен обзор сетевых компонентов Windows, показано, как каждый компонент вписывается в исходную модель OSI и какие протоколы используются между уровнями. Сопоставление OSI-уровней и сетевых компонентов не может быть точным, поскольку некоторые компоненты относятся к нескольким уровням. В состав разнообразных компонентов входят следующие:

- *Сетевые API-интерфейсы*, предоставляющие независимый от протокола способ, позволяющий приложениям обмениваться данными по сети. Сетевые API-интерфейсы могут быть реализованы в пользовательском режиме или как в пользовательском режиме, так и в режиме ядра. В некоторых случаях они служат оболочкой для других сетевых API-интерфейсов, реализующих определенную модель программирования или предоставляющих дополнительные виды обслуживания. (Следует иметь в виду, что под понятие *сетевого API-интерфейса* подпадают любые интерфейсы программирования интерфейса, предоставляемые программным обеспечением, связанным с работой в сети.)
- *Клиенты интерфейса транспортного драйвера — Transport Driver Interface (TDI)*, являющиеся унаследованными драйверами устройств режима ядра, которые обычно реализуют ту часть сетевого API-интерфейса, которая работает в режиме ядра. TDI-клиенты получили свое название на том основании, что отправляемые ими драйверам протоколов пакеты запросов на ввод-вывод — I/O request packets (IRP) — отформатированы в соответствии со стандартом Windows Transport Driver Interface (документация по этому стандарту находится в Windows Driver Kit). В этом стандарте указывается общий интерфейс программирования для драйверов устройств режима ядра. Интерфейс TDI не рекомендован к использованию и из будущих версий Windows он будет удален. Сейчас TDI-интерфейс экспортируется драйвером TDI Extension (TDX). В настоящее время клиенты сети, работающие в режиме ядра, должны для обращения к сетевому стеку использовать интерфейс Winsock Kernel (WSK).
- *TDI-транспорты* (также известные как *транспорты*) и драйверы протокола Network Driver Interface Specification (NDIS) (или просто драйверы протокола), являющиеся драйверами сетевого протокола режима ядра. Они принимают IRP-пакеты от TDI-клиентов и обрабатывают запросы, представленные этими IRP-пакетами. Эта обработка может потребовать обмен данными по сети с равным по уровню узлом, побуждая TDI-транспорт добавлять специфичные для протокола заголовки (например, TCP, UDP и (или) IP) к данным, передаваемым в IRP, и обмениваться данными с драйверами адаптера с помощью NDIS-функций (описание этих функций дается в Windows Driver Kit). TDI-транспорты, как правило, облегчают связь сетевых приложений путем прозрачного проведения операций по отправке сообщений, например о сегментации и повторной сборке, установлении последовательности, подтверждении и о повторной передаче.

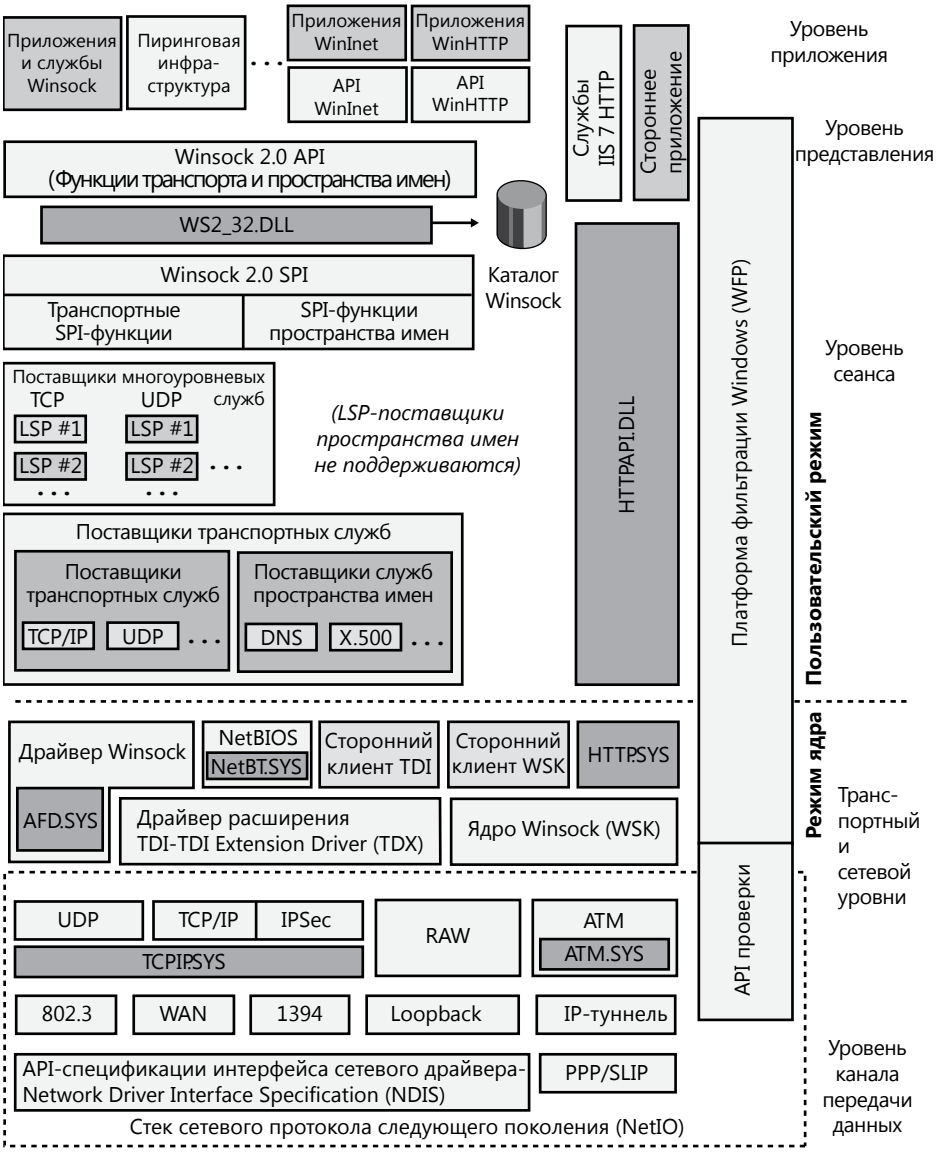


Рис. 7.2. Модель OSI и сетевые компоненты Windows

- В компании Microsoft решили, что TCP/IP победил в войне сетевых протоколов, поэтому там была проведена переработка той части сетевого стека, которая относится к сетевому протоколу, превратив ее из нейтральной к протоколу к строго ориентированной на TCP/IP. Интерфейс между драйвером протокола TCP/IP и Winsock известен как *интерфейс предоставления транспортного сетевого уровня — Transport Layer Network Provider Interface (TLNPI)*, и в настоящее время документация на него не составлена.

- *Ядро Winsock — Winsock Kernel (WSK)*, являющийся независимым от транспортов сетевым API-интерфейсом режима ядра, заменившим устаревший TDI. WSK предоставляет сетевой обмен данными путем использования семантики программирования, похожей на сокеты, аналогичную Winsock пользовательского режима, наряду с предоставлением таких уникальных свойств, как асинхронные операции ввода-вывода, основанные на пакетах запросов ввода-вывода (IRP) и на функциях обратного вызова, связанных с событиями. WSK также изначально поддерживает функциональность IP версии 6 (IPv6) в сетевом стеке нового поколения — Next Generation TCP/IP в Windows.
- *Платформа фильтрации Windows — Windows Filtering Platform (WFP)*, являющаяся набором API-функций и системных служб, предоставляющих возможность создания сетевых фильтрующих приложений. WFP позволяет приложениям взаимодействовать с пакетной обработкой на разных уровнях сетевого стека Windows, что очень похоже на фильтры файловой системы. Аналогичным образом сетевые данные могут подвергаться трассировке, фильтрации, а также изменяться до достижения пункта назначения.
- *Вызывающие драйверы WFP — WFP callout drivers*, являющиеся драйверами режима ядра, реализующими один или несколько вызовов, которые расширяют возможности WFP путем обработки сетевых данных, основанных на протоколе TCP/IP такими способами, которые расширяют основные функциональные возможности, предоставляемые WFP.
- *Библиотека NDIS (Ndis.sys)*, предоставляющая механизм абстракции, инкапсулирующий драйверы сетевого адаптера (Network Interface Card, NIC), известный также как мини-порты спецификации стандартного интерфейса сетевых адаптеров (NDIS miniports), который скрывает от них особенности Windows-среды режима ядра. NDIS-библиотека экспортирует функции для использования транспорта TCP/IP и устаревшего транспорта TDI.
- *Драйверы мини-порта NDIS*, являющиеся драйверами режима ядра, отвечающими за согласование сетевого стека с конкретным NIC. Драйверы мини-порта NDIS написаны таким образом, что они заключены в оболочку NDIS-библиотеки Windows. Драйверы мини-порта NDIS не обрабатывают IRP-пакеты, вместо этого они регистрируют интерфейс таблицы вызовов к библиотеке NDIS, который содержит указатели на функции, выполняющие простые операции на NIC, например отправка пакета или запрос свойств. Драйверы мини-порта NDIS обмениваются данными с сетевыми адаптерами, используя функции библиотеки NDIS, которые разделяются на функции уровня аппаратных абстракций — hardware abstraction layer (HAL).

На рис. 7.2 показано, что OSI-уровни не согласуются с фактическим программным обеспечением. Например, поставщики транспорта WSK часто переходят сразу несколько границ. Фактически самые нижние три уровня программного обеспечения и нижний уровень аппаратного обеспечения часто носят общее название *транспорта*. Программные компоненты, которые находятся в верхних трех уровнях, называются пользователями или клиентами транспорта.

Далее в этой главе будут исследованы сетевые компоненты, показанные на рис. 7.2 (а также другие компоненты, не показанные на рисунке), их сопряжение и отношение ко всей системе Windows.

Сетевые API

В Windows реализованы несколько API-интерфейсов, обеспечивающих поддержку устаревших приложений и совместимость с промышленными стандартами. В данном разделе будет дан краткий обзор сетевых API и описание порядка их использования приложениями. Следует иметь в виду, что решение о том, какой API следует использовать приложению, зависит от характеристик API, например, какие протоколы могут накладываться API, поддерживает ли API надежный (или двунаправленный) обмен данными и переносим ли API на другие платформы Windows, на которых может запускаться приложение. Мы рассмотрим следующие сетевые API:

- сокеты Windows — Windows Sockets (Winsock);
- ядро Winsock — Winsock Kernel (WSK);
- удаленный вызов процедур — Remote procedure call (RPC);
- API-интерфейсы доступа к Интернету — Web access API;
- именованные каналы (named pipes) и почтовые слоты (mailslots);
- NetBIOS;
- другие сетевые API.

Сокеты Windows

Исходные Windows-сокеты (Winsock) версии 1.0 были созданы компанией Microsoft в качестве реализации сокетов BSD (Berkeley Software Distribution), программного API, превратившегося с 1980-х годов в стандарт обмена данными между UNIX-системами через Интернет. Поддержка сокетов в Windows превращает перенос сетевых приложений UNIX в Windows в относительно простую задачу. Современные версии Winsock включают большинство функциональных возможностей BSD-сокетов, но также включают усовершенствования, характерные для программного обеспечения Microsoft-specific, которые продолжают совершенствоваться. Winsock поддерживают надежную, сориентированную на соединение связь, а также ненадежную связь, не использующую соединения[1]. Windows предоставляет сокеты Winsock 2.2, к которым добавлен ряд свойств, принадлежащих спецификации BSD Sockets, например функции, использующие асинхронный ввод-вывод Windows, чтобы предложить намного более высокую производительность и масштабируемость, чем этого можно добиться с помощью программирования с непосредственным использованием BSD Sockets.

Winsock включают следующие свойства:

- Поддержку приложений ввода-вывода, работающих по принципу разброса-сбора (scatter-gather), и приложений, работающих в асинхронном режиме.
- Соглашение по качеству обслуживания — Quality of Service (QoS), чтобы приложения могли согласовать требования ко времени ожидания и к пропускной способности, когда сеть, через которую они работают, поддерживает QoS.

[1] О «надежности» здесь говорится в том смысле, что отправитель уведомляется о любых проблемах, возникающих при доставке данных получателю.

- Возможность расширения, чтобы Winsock мог использоваться с протоколами сторонних разработчиков (не рекомендуется).
- Поддержку интегрированного пространства имен со сторонними поставщиками пространств имен. Сервер, например, может опубликовать свое имя в Active Directory, и, используя расширения пространства имен, клиент может найти адрес сервера в Active Directory.
- Поддержку многоадресных сообщений, где сообщения передаются из одного источника нескольким получателям.

Мы изучим типовые операцииWinsock, а затем рассмотрим способы, с помощью которых может быть расширен Winsock.

Работа клиента Winsock

В первую очередь приложение Winsock инициализирует Winsock API с помощью вызова функции инициализации. Следующим действием приложения Winsock является создание сокета, который будет представлять конечную точку связи. Приложение получает адрес сервера, к которому оно хочет подключиться путем вызова функции getaddrinfo (с последующим вызовом freeaddrinfo, чтобы передать информацию). Функция getaddrinfo возвращает список адресов, имеющих отношение к протоколу, назначенных серверу, и клиент пытается подключиться к каждому из них по очереди, пока он не сможет установить подключение к одному из них. Тем самым гарантируется, что клиент, поддерживающий как IP версии 4 (IPv4), так и IPv6 подключится к соответствующему и (или) наиболее эффективному адресу сервера, который может располагать назначенными ему адресами как IPv4, так и IPv6. (IPv6, предпочтительнее IPv4.) Winsock является независимым от протокола API-интерфейсом, поэтому адрес может быть указан для любого протокола, установленного в системе, через который работает Winsock. После получения адреса сервера клиент, ориентированный на установку соединения, пытается подключиться к серверу, используя функцию connect и указывая адрес сервера.

Когда подключение установлено, клиент может отправлять и получать данные через свой сокет, используя API-функции recv и send. Клиент, не ориентированный на установку соединения (connectionless), указывает удаленный адрес с помощью API-функций, не рассчитанных на установку соединения, например с помощью эквивалентов send и recv, по имени sendto и recvfrom. Клиенты могут также использовать API-функции select и WSAPoll для ожидания или опроса нескольких сокетов для синхронных операций ввода-вывода или для проверки их состояния.

Работа сервера Winsock

Последовательность действий серверного приложения отличается от последовательности действий на стороне клиента. После инициализации Winsock API сервер создает сокет, а затем привязывает его к локальному адресу, используя функцию bind. Здесь также указывается адресное семейство: будет ли это TCP/IPv4, TCP/IPv6 или какое-нибудь другое адресное семейство, зависит от серверного приложения.

Если сервер сориентирован на установку соединения, он выполняет в отношении сокета операцию прослушивания — listen, указывая запас, или количество соединений, которое сервер просит Winsock удерживать, пока он не сможет их принять. Затем он выполняет операцию приема — accept, чтобы позволить клиенту подключиться к сокету. Если запрашивается ожидающее соединение, вызов функции accept тут же завершается, в противном случае он завершается, когда прибывает запрос на подключение. Когда соединение установлено, функция accept возвращает новый сокет, который представляет собой серверный конечный пункт соединения. (Исходный сокет, используемый для прослушивания, для установки связей не используется, он используется только для получения запросов на подключения.) Сервер может выполнять операции получения и отправки путем использования таких функций, как recv и send. Подобно клиентам Winsock, серверы могут использовать функции select и WSAPoll для запроса состояния одного или нескольких сокетов, но для лучшей масштабируемости предпочтительнее пользоваться Winsock-функцией WSAEventSelect и совмещенными (асинхронными) расширениями ввода-вывода. На рис. 7.3 показана связь, ориентированная на установку соединения между клиентом и сервером.

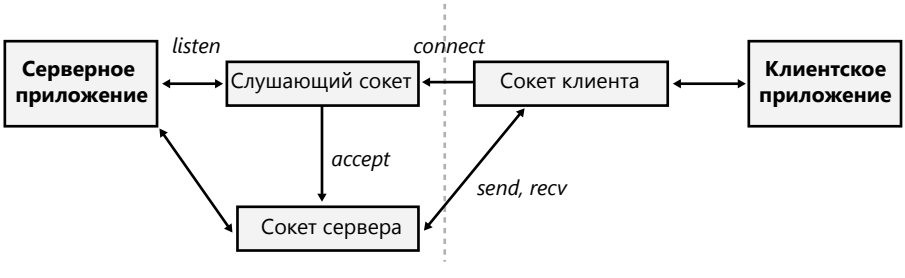


Рис. 7.3. Операции Winsock, ориентированные на установку соединения

После привязки адреса сервер, не сориентированный на установку соединения, не отличается от клиента, не сориентированного на установку соединения: он может отправлять и получать данные через сокет, просто указав адрес с каждой операцией. Большинство протоколов, не ориентированных на установку соединения, считаются ненадежными и, в общем, не будут знать, получили ли место назначения отправленные пакеты данных (которые известны как датаграммы). Протоколы датаграмм идеально подходят для передачи быстрых сообщений там, где издержки при установке соединения слишком велики и особой надежности не требуется (хотя приложение может добиться надежности в верхней части протокола).

Расширения Winsock

Кроме поддержки функций, которые напрямую соответствуют функциям, реализованным в BSD Sockets, Microsoft добавила небольшое количество функций, не являющихся частью стандарта BSD. Две из этих функций, AcceptEx и TransmitFile, стоят того, чтобы их рассмотреть, потому что многие веб-серверы Windows используют их для достижения высокой производительности. Функция AcceptEx является версией функции accept, которая в процессе установки соединения

с клиентом возвращает адрес клиента и первое сообщение клиента. AcceptEx позволяет серверному приложению ставить в очередь несколько операций accept, чтобы можно было обработать большие объемы входящих запросов на подключение. С помощью этой функции веб-серверу удается избежать выполнения сразу нескольких Winsock-функций, без чего иначе было бы не обойтись.

После установки соединения с клиентом веб-сервер часто отправляет клиенту файлы, например веб-страницы. Реализация функции TransmitFile интегрирована с диспетчером кэша Windows, поэтому файлы могут отправляться непосредственно из кэша файловой системы. Отправка данных таким способом называется нулевым копированием (zero-copy), потому что серверу не нужно читать данные файла для их отправки, он просто указывает для отправки дескриптор файла и диапазон байтов (смещение и длину) файла. Кроме того, функция TransmitFile позволяет серверу добавлять к данным файла данные префикса или суффикса, чтобы сервер мог отправить заголовочную информацию, замыкающую информацию или и ту и другую, которая может включать в себя имя веб-сервера и поле, показывающее клиенту размер отправленного сервером сообщения. Информационные службы Интернета — Internet Information Services (IIS), включенные в Windows, для достижения лучшей производительности используют обе функции, и AcceptEx, и TransmitFile.

Windows также поддерживает небольшое количество других многофункциональных API, включая ConnectEx, DisconnectEx и TransmitPackets. Функция ConnectEx устанавливает подключение и отправляет первое сообщение по этому подключению. Функция DisconnectEx закрывает подключение и позволяет дескриптору сокета представлять заново используемое подключение в вызовах функции AcceptEx или функции ConnectEx. И наконец, функция TransmitPackets похожа на функцию TransmitFile, за исключением того, что она позволяет отправлять данные, находящиеся в памяти, в дополнение к данным файла или вместо них. В конечном счете, используя функции WSAImpersonateSocketPeer и WSARevertImpersonation, серверы Winsock могут выполнять заимствование прав доступа (см. главу 6) для выполнения авторизации или для получения доступа к ресурсам на основе учетных данных безопасности клиента.

Практическое расширение Winsock

Winsock является в Windows расширяемым API, поскольку сторонние разработчики могут добавить поставщика транспортной службы, который организует интерфейс Winsock с другими протоколами, или уровни, являющиеся надстройками над существующими протоколами, чтобы предоставить такие функциональные возможности, как организация прокси-серверов. Сторонние разработчики могут также добавить поставщика службы пространства имен для пополнения имеющихся в распоряжении Winsock средств разрешения имен. Поставщики служб подключаются к Winsock с помощью имеющегося в Winsock интерфейса поставщиков служб — service provider interface (SPI). Когда поставщик транспортной службы регистрируется с помощью Winsock, служба Winsock использует поставщика транспортной службы для реализации функций сокета, например connect и accept, для тех типов адресов, которые поставщик указал в качестве им реализуемых. Ограничений на то, как поставщик транспортной службы реализует

функции, не накладывается, но реализация обычно использует обмен данными с транспортным драйвером в режиме ядра.

ПРИМЕЧАНИЕ

Поставщики многоуровневых служб небезопасны и могут быть обойдены; надстройки над безопасным сетевым протоколом должны выполняться в режиме ядра. Технология установки самих себя в качестве поставщика многоуровневой службы для Winsock — Winsock layered service provider (LSP) — часто используется вредоносными и шпионскими программами.

От любого клиент-серверного приложения Winsock требуется, чтобы сервер сделал свои адреса доступными клиентам, чтобы клиенты могли подключиться к серверу. Стандартные серверы, работающие на основе протокола TCP/IP, используют широко известные адреса, чтобы сделать свои адреса доступными. Если браузер знает имя компьютера, на котором запущен веб-сервер, он может подключиться к веб-серверу, указав широко известный адрес веб-сервера (IP-адрес сервера соединенный со строковым значением :80, номером порта, используемого HTTP). Поставщики служб пространства имен дают серверам возможность зарегистрировать свое присутствие другими способами. Например, один поставщик службы пространства имен может на серверной стороне зарегистрировать серверный адрес в Active Directory, а на клиентской стороне найти серверный адрес в Active Directory. Поставщики службы пространства имен предоставляют эту функциональную возможность Winsock путем реализации стандартных Winsock-функций разрешения имен, таких как getaddrinfo и getnameinfo.

ЭКСПЕРИМЕНТ: ПРОСМОТР СЛУЖБЫ WINSOCK И ПОСТАВЩИКОВ ПРОСТРАНСТВА ИМЕН

Утилита Network Shell (Netsh.exe), входящая в состав Windows, с помощью команды netsh winsock show catalog может показать зарегистрированный транспорт Winsock и поставщиков пространства имен. Например, если есть два поставщика транспортных служб TCP/IP, то первым показанным будет исходный поставщик для приложений Winsock, использующий протокол TCP/IP. Вот как выглядит пример вывода утилиты Netsh, в котором показаны зарегистрированные поставщики транспортных служб:

```
C:\Users\Toby>netsh winsock show catalog
Winsock Catalog Provider Entry
-----------------------------------------------------
Entry Type: Base Service Provider
Description: MSAFD Tcpip [TCP/IP]
Provider ID: {E70F1AA0-AB8B-11CF-8CA3-00805F48A192}
Provider Path: %SystemRoot%\system32\mswsock.dll
Catalog Entry ID: 1001
Version: 2
Address Family: 2
Max Address Length: 16
Min Address Length: 16
Socket Type: 1
Protocol: 6
Service Flags: 0x20066
```

```
Protocol Chain Length: 1
Winsock Catalog Provider Entry
----------------------------------------------------
Entry Type: Base Service Provider
Description: MSAFD Tcpip [UDP/IP]
Provider ID: {E70F1AA0-AB8B-11CF-8CA3-00805F48A192}
Provider Path: %SystemRoot%\system32\mswsock.dll
Catalog Entry ID: 1002
Version: 2
Address Family: 2
Max Address Length: 16
Min Address Length: 16
Socket Type: 2
Protocol: 17
Service Flags: 0x20609
Protocol Chain Length: 1
Winsock Catalog Provider Entry
----------------------------------------------------
Entry Type: Base Service Provider
Description: MSAFD Tcpip [RAW/IP]
Provider ID: {E70F1AA0-AB8B-11CF-8CA3-00805F48A192}
Provider Path: %SystemRoot%\system32\mswsock.dll
Catalog Entry ID: 1003
Version: 2
Address Family: 2
Max Address Length: 16
Min Address Length: 16
Socket Type: 3
Protocol: 0
Service Flags: 0x20609
Protocol Chain Length: 1
.
.
.
Name Space Provider Entry
----------------------------------------------------
Description: Network Location Awareness Legacy (NLAv1) Namespace
Provider ID: {6642243A-3BA8-4AA6-BAA5-2E0BD71FDD83}
Name Space: 15
Active: 1
Version: 0
Name Space Provider Entry
----------------------------------------------------
Description: E-mail Naming Shim Provider
Provider ID: {964ACBA2-B2BC-40EB-8C6A-A6DB40161CAE}
Name Space: 37
Active: 1
Version: 0
Name Space Provider Entry
----------------------------------------------------
```

продолжение ⇩

```
Description: PNRP Cloud Namespace Provider
Provider ID: {03FE89CE-766D-4976-B9C1-BB9BC42C7B4D}
Name Space: 39
Active: 1
Version: 0
.
.
.
```

Для просмотра пространства имен и транспортных поставщиков, а также для выключения или удаления тех из них, которые могут создавать проблемы или вести себя в системе нежелательным образом, можно воспользоваться утилитой Autoruns из набора Windows Sysinternals (www.microsoft.com/technet/sysinternals). ■

Реализация Winsock

Реализация Winsock показана на рис. 7.4. Его прикладной интерфейс состоит из API DLL, Ws2_32.dll (%SystemRoot%\System32\Ws2_32.dll), предоставляющего приложениям доступ к функциям Winsock Ws2_32.dll службы пространства имен и поставщику транспортных служб для выполнения операций, связанных с именами и сообщениями. Библиотека Mswsock.dll (%SystemRoot%\System32\mswsock.dll) работает в качестве поставщика транспортной службы для протоколов, поддерживаемых Microsoft, и использует ориентированные на конкретные протоколы библиотеки Winsock Helper для связи с драйверами протоколов режима ядра. Например, Wshtcpip.dll (%SystemRoot%\System32\wshtcpip.dll) является вспомогательной библиотекой TCP/IP. Библиотека Mswsock.dll реализует расширенные функции Microsoft Winsock, например TransmitFile и WSARecvEx.

Windows поставляется со вспомогательными DLL-библиотеками TCP/IPv4, TCPv6, Bluetooth, NetBIOS, IrDA (Infrared Data Association — ассоциация по передаче данных по инфракрасному каналу) и протоколом надежной многоадресной рассылки PGM (Pragmatic General Multicast). Она также включает поставщиков пространства имен для DNS (TCP/IP), Active Directory (NTDS), NLA (Network Location Awareness), PNRP (Peer Name Resolution Protocol) и Bluetooth.

Подобно API-интерфейсам именованных каналов и почтовых слотов (рассматриваемых далее в этой главе), Winsock для представления сокетов интегрируется с моделью ввода-вывода Windows и использует дескрипторы файлов. Эта поддержка требует помощи драйвера режима ядра, поэтому библиотека Msafd.dll (%SystemRoot%\System32\msafd.dll) для реализации основанных на сокетах функций использует службы драйвера вспомогательных функций — Ancillary Function Driver (AFD — %SystemRoot%\System32\Drivers\Afd.sys). AFD является клиентом поставщика интерфейса сетевого транспортного уровня — Transport Layer Network Provider Interface (TLNPI) — и выполняет операции сетевого сокета, например отправку и получение сообщений. TLNPI является недокументированным интерфейсом между стеком протоколов AFD и TCP/IP. Если установлен драйвер устаревшего протокола, Windows будет использовать TDI-TLNPI, драйвер трансляции TDX (%SystemRoot%\System32\Drivers\tdx.sys) для отображения TDI IRP-пакетов на TLNPI-запросы.

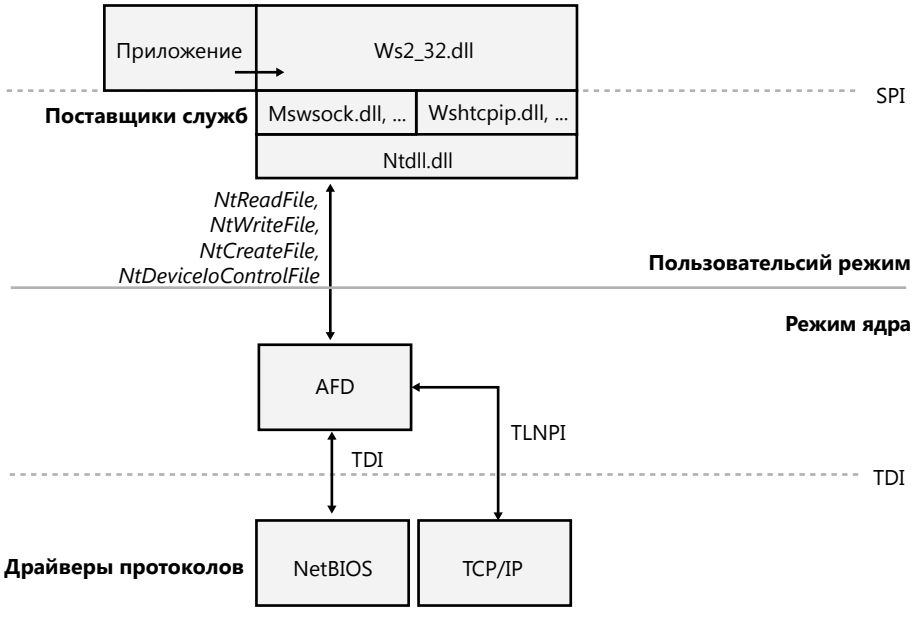


Рис. 7.4. Реализация Winsock

Ядро Winsock

Чтобы разрешить драйверам режима ядра и модулям иметь доступ к сетевым API-интерфейсам, подобным тем, которые доступны приложениям пользовательского режима, Windows реализует основанный на сокетах интерфейс сетевого программирования, который называется Winsock Kernel (WSK). WSK заменяет устаревший интерфейс TDI API, присутствующий на более старых версиях Windows, но сохраняет этот интерфейс для транспортных поставщиков. По сравнению с TDI, WSK предоставляет лучшую производительность, лучшую масштабируемость и более простую парадигму программирования, поскольку он меньше полагается на внутреннее поведение ядра и больше использует семантику на основе сокетов. Кроме того, WSK была написана для максимального использования самых последних технологий в стеке Windows TCP/IP, поддержка которых в TDI изначально не предусматривалась. Как показано на рис. 7.5, для подключения к транспортным протоколам и отключения от них WSK использует Windows-компонент регистрации сетевых модулей — Network Module Registrar (NMR), являющийся частью %SystemRoot%\System32\drivers\NetIO.sys, — который может быть использован точно так же, как Winsock, для поддержки множества типов сетевых клиентов. Например, драйвер Http.sys для HTTP Server API (который рассматривается далее в этой главе) является WSK-клиентом. Использование NMR с WSK дается непросто, поэтому API-функции, поддерживающие регистрацию, предоставляются для регистрации вместе с WSK (WskRegister, WskDeregister, WskCaptureProviderNPI и WskReleaseProviderNPI).

ПРИМЕЧАНИЕ

Транспортный протокол Raw не является настоящим протоколом и не осуществляет никакой инкапсуляции пользовательских данных. Это позволяет клиенту непосредственно контролировать содержимое фреймов, отправляемых и получаемых по сетевому интерфейсу.

WSK улучшает безопасность, ограничивая распределение адресов посредством использования нестандартного распределения и дескрипторов безопасности адресов. Это позволяет нескольким сокетам использовать один и тот же транспортный (TCP/IP) адрес. WSK использует для адреса дескриптор безопасности, определенный первым сокетом, и проверяет владеющий процесс и поток при каждой последующей попытке получить этот адрес.

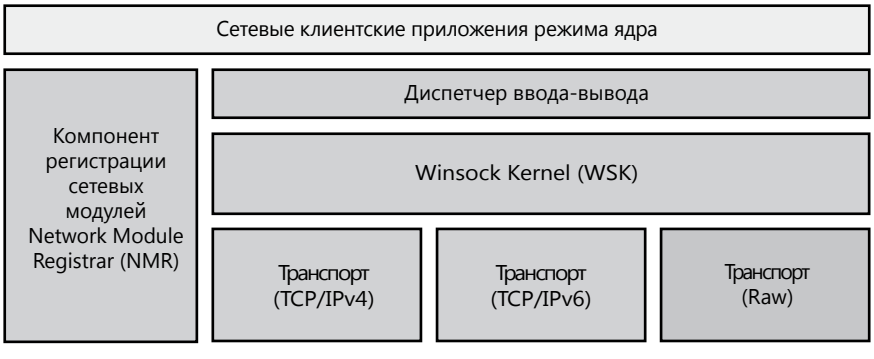


Рис. 7.5. Общий вид WSK

Реализация WSK

Реализация WSK показана на рис. 7.6. В его основу положена сама подсистема WSK, которая использует стек TCP/IP следующего поколения — Next Generation TCP/IP Stack (%SystemRoot%\System32\Drivers\Tcpip.sys) и вспомогательную библиотеку NetIO (%SystemRoot%\System32\Drivers\NetIO.sys), но на самом деле WSK реализовано в AFD. Подсистема отвечает за сторону поставщиков WSK API. Подсистема сопрягается с транспортными протоколами TCP/IP (показанными в нижней части рис. 7.5). Присоединенные к WSK подсистемы являются клиентами WSK, к которым относятся драйверы режима ядра, реализующие WSK API клиентской стороны с целью выполнения сетевых операций. Подсистема WSK вызывает клиентов WSK, чтобы уведомить их об асинхронных событиях.

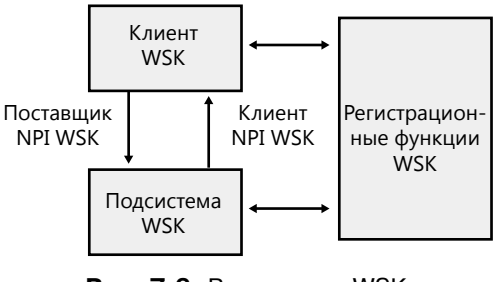


Рис. 7.6. Реализация WSK

Клиенты WSK привязаны к подсистеме WSK через NMR или через регистрационные функции WSK, позволяющие клиентам WSK динамически обнаруживать, когда подсистема WSK становится доступной, а затем загружать свою собственную таблицу переходов для описания поставщика и реализации WSK API на стороне клиента. Эти реализации предоставляют стандартные WSK-функции на основе сокетов, такие как WskSocket, WskAccept, WskBind, WskConnect, WskReceive и WskSend, которые имеют похожую семантику (но не обязательно похожие параметры) с их Winsock-аналогами в пользовательском пространстве. Но, в отличие от Winsock пользовательского режима, подсистема WSK определяет четыре разновидности категорий сокетов, которые определяют, какие функции и события доступны:

- Базовые сокеты, используемые только для получения и установки информации о транспорте. Они не могут использоваться для отправки или получения данных или для привязки к адресу.
- Слушающие сокеты, используемые только для принятия входящих подключений.
- Сокеты датаграмм, использующиеся исключительно для отправки и получения датаграмм.
- Сокеты, сориентированные на образование соединения, которые поддерживают все функции, требуемые для отправки и получения сетевого трафика через установленное соединение.

Кроме рассмотренных функций сокетов, WSK также предоставляет события, посредством которых клиенты уведомляются о состоянии сети. В отличие от модели для функций сокетов, в которой подключением управляет клиент, события позволяют подсистеме управлять подключениями и просто уведомлять клиента. В число управляющих процедур входят WskAcceptEvent, WskInspectEvent, WskAbortEvent, WskReceiveFromEvent, WskReceiveEvent, WskDisconnectEvent и WskSendBacklogEvent.

И наконец, как и Winsock пользовательского режима, WSK может быть расширено через интерфейсы расширения, которые клиенты могут связать с сокетами. Эти расширения могут улучшить исходную функциональность, предоставляемую подсистемой WSK.

Вызов удаленной процедуры

Вызов удаленной процедуры — Remote procedure call (RPC) — является стандартом сетевого программирования, который изначально был разработан в начале 1980-х гг. Фонд открытого программного обеспечения Open Software Foundation (теперь Open Group) сделал RPC частью распределенной среды вычислений — distributed computing environment (DCE), — являющейся стандартом распределенного вычисления. Хотя есть еще и второй RPC-стандарт под названием SunRPC, реализация Microsoft RPC совместима со стандартом OSF/DCE. RPC строится на других сетевых API-интерфейсах, таких как именованные каналы или Winsock, для предоставления альтернативной модели программирования, которая в некотором отношении скрывает детали сетевого программирования от разработчика приложения.

По сути, RPC предоставляет механизм для создания программ, распространяемых по сети, с дозой приложений, выполняемых явным образом на одной или нескольких системах.

Работа RPC

RPC является одним из средств, позволяющих программисту создавать приложение, состоящее из любого количества процедур, часть из которых выполняется локально, а другая часть выполняется на удаленном компьютере через сеть. Он предоставляет процедурный взгляд на сетевые операции, а не взгляд, сконцентрированный на транспортировку данных, упрощая тем самым создание распределенных приложений.

Сетевое программное обеспечение традиционно структурируется вокруг модели обработки ввода-вывода. Например, в Windows сетевая операция инициируется, когда приложение выдает запрос на ввод-вывод. Операционная система обрабатывает запрос соответствующим образом, пересылая его *редиректору*, который работает как удаленная файловая система, делая взаимодействие клиента с удаленной файловой системой невидимым для клиента. Редиректор передает операцию удаленной файловой системе, и после того, как удаленная система выполнит запрос и вернет результаты, локальная сетевая карта выдает прерывание. Ядро обрабатывает прерывание, и исходная операция ввода-вывода завершается, возвращая результаты вызывающему процессу.

При RPC-вызове применяется совершенно другой подход. RPC-приложения похожи на другие структурированные приложения с основной программой, вызывающей для выполнения определенных задач процедуры или библиотеки процедур. Разница между RPC-приложениями и обычными приложениями состоит в том, что некоторые библиотеки процедур в RPC-приложениях хранятся и выполняются на удаленных компьютерах, как показано на рис. 7.7, а другие выполняются локально.

Для RPC-приложения все процедуры предстают в качестве локально выполняемых. Иными словами, вместо того чтобы вынуждать программиста создавать код для передачи по сети вычислительных запросов или запросов на ввод-вывод, заниматься обработкой сетевых протоколов, разбираться с ошибками сети и т. д., программное обеспечение RPC справляется с этими задачами автоматически. И имеющиеся в Windows средства RPC могут работать через любые доступные транспортные протоколы, загруженные в систему.

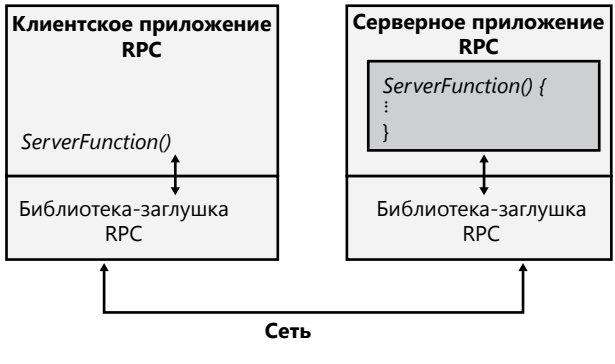


Рис. 7.7. Работа RPC

Чтобы написать RPC-приложение, программист решает, какие процедуры будут выполняться локально, а какие удаленно. Предположим, к примеру, что обычная рабочая станция подключена к суперкомпьютеру (машина с очень высоким быстродействием, разработанная для высокоскоростных векторных операций). Если бы программист создавал приложение, работающее с большими матрицами, то с точки зрения производительности был бы смысл переложить математические вычисления на суперкомпьютер, написав программу в виде RPC-приложения.

RPC-приложения работают следующим образом. При запуске приложения оно вызывает локальные процедуры, а также процедуры, которые не присутствуют на локальной машине. Для обработки последнего случая приложение связано с локальной библиотекой или DLL-библиотекой, в которой содержатся *процедуры-заглушки*, по одной на каждую удаленную процедуру. Чтобы упростить приложения, процедуры-заглушки статически связаны с приложением, но для более крупных компонентов заглушки включаются в отдельные DLL-библиотеки. В модели DCOM, рассматриваемой в данной главе чуть позже, обычно используется последний метод. Процедуры-заглушки имеют такие же имена и используют такой же интерфейс, как и удаленные процедуры, но вместо выполнения требуемых операций заглушка берет переданные ей параметры и *маршализирует* их для передачи по сети. Маршализирование параметров означает их выстраивание и пакетирование определенным способом с учетом сетевого соединения, например с разрешением ссылок и с подбором копии любых структур данных, на которые ссылается указатель.

Затем процедура-заглушка вызывает библиотеку RPC времени выполнения, которая определяет местонахождение компьютера, на котором находится удаленная процедура, определяет, какой сетевой транспортный механизм этот компьютер использует, и отправляет ему запрос, используя локальное транспортное программное обеспечение. Когда удаленный сервер получает RPC-запрос, он проводит *демаршализацию* параметров (выполняет операцию, обратную маршализации), реконструирует исходный вызов процедуры и вызывает процедуру с параметрами, переданными из вызывающей системы. Когда сервер завершает выполнение процедуры, он проделывает все в обратном порядке, чтобы вернуть результаты вызывающей процедуре.

Кроме рассмотренного здесь синхронного интерфейса, основанного на вызове функции, Windows RPC также поддерживает *асинхронный RPC*, который позволяет RPC-приложению выполнять функцию, но не дожидаться, пока она завершится, чтобы продолжить свою работу. Вместо этого приложение может выполнять другой код, а чуть позже, когда от сервера придет ответ, библиотека RPC времени выполнения уведомляет клиента о завершении операции. Библиотека RPC времени выполнения использует уведомительный механизм, запрошенный клиентом. Если клиент использует для уведомления объект синхронизации события, она ждет сигнала от объекта события, вызывая либо функцию WaitForSingleObject, либо функцию WaitForMultipleObjects. Если клиент предоставляет асинхронный вызов процедуры — asynchronous procedure call (APC), — библиотека времени выполнения выстраивает очередь выполнения APC для потока, который выполняет функцию RPC[1]. Если клиентская программа использует в качестве механизма уведомления порт завершения ввода-вывода,

[1] APC не будет доставлен, пока запрашиваемый поток не войдет в *состояние ожидания извещения*. Дополнительные сведения о APC даны в главе 3 «Системные механизмы».

она должна вызвать функцию GetQueuedCompletionStatus, чтобы узнать о завершении работы функции. Кроме того, клиент может проводить опрос завершения, вызвав функцию RpcAsyncGetCallStatus.

Кроме библиотеки RPC времени выполнения имеющиеся в Microsoft средства обслуживания RPC включают компилятор, который называется компилятором языка определения интерфейса Microsoft — *Microsoft Interface Definition Language* (MIDL). MIDL-компилятор упрощает создание RPC-приложений путем генерирования необходимых процедур-заглушек. Программист пишет серию обычных прототипов функций (применительно к приложениям на языке C или C++), которые описывают удаленные процедуры, а затем помещает процедуры в файл. Затем программист добавляет к этим прототипам дополнительную информацию, например уникальный идентификатор сети для пакета процедур и номер версии, плюс атрибуты, определяющие, являются ли параметры входными, выходными или и теми и другими. Приукрашенные прототипы формируют принадлежащий разработчику файл языка определения интерфейса — Interface Definition Language (IDL).

После создания IDL-файла программист компилирует его с помощью MIDL-компилятора, который создает процедуры-заглушки на стороне клиента и на стороне сервера, а также заголовочные файлы, которые нужно включить в приложение. Когда приложение на стороне клиента привязывается к файлу процедур-заглушек, разрешаются все ссылки удаленных процедур. Затем устанавливаются удаленные процедуры, для чего используются аналогичные процессы на серверной машине. Программист, желающий вызвать существующее RPC-приложение, должен лишь написать клиентскую часть программного обеспечения и связать приложение с локальным средством обслуживания библиотеки RPC времени выполнения.

Библиотекой RPC времени выполнения для общения с транспортным протоколом используется *универсальный транспортный поставщик* RPC. Интерфейс поставщика действует, как тонкий уровень между средством RPC и транспортом, отображая RPC-операции на функции, предоставляемые транспортом. Имеющееся в Windows средство обслуживания RPC реализует DLL-библиотеки транспортного поставщика для именованных каналов, HTTP, TCP/IP и UDP. Подобным же образом, средство RPC рассчитано на работу с различными средствами сетевой безопасности.

Большинство сетевых служб Windows является RPC-приложениями, а это значит, что их могут вызывать как локальные приложения, так и приложения на удаленных компьютерах. Таким образом, удаленный клиентский компьютер может вызывать серверную службу для получения списка общих ресурсов, открытия файлов, записи в очередь вывода на печать или для активации пользователей на вашем сервере, конечно же, все это в условиях ограничений, накладываемых системой безопасности. Основной объем API-функций, занимающихся управлением клиентами, реализован с использованием RPC.

Публикация имени сервера, которая является возможностью сервера зарегистрировать его имя в месте, доступном для просмотра клиентами, реализуется в RPC и интегрируется с Active Directory. Если компонент Active Directory не установлен, службы локатора имен RPC прибегают к рассылке с помощью NetBIOS. Такое поведение позволяет RPC функционировать на автономных серверах и рабочих станциях.

Обеспечение безопасности RPC

Windows RPC включает интеграцию с поставщиками поддержки безопасности —security support providers (SSP), — чтобы RPC клиенты и серверы могли использовать аутентификацию или зашифрованную связь. Когда RPC сервер хочет получить зашифрованную связь, он сообщает об этом библиотеке времени выполнения RPC, какую службу аутентификации добавить с списку доступных *служб аутентификации*. Когда клиент хочет воспользоваться безопасной связью, он связывается с сервером. На этот раз он должен сообщить библиотеке RPC времени выполнения о желаемой службе аутентификации и о требуемом *уровне аутентификации*. Существуют различные уровни аутентификации для обеспечения подключения к серверу только авторизованных клиентов, проверки, что каждое сообщение, получаемое сервером, исходит от авторизованного клиента, проверки целостности RPC-сообщений с целью обнаружения манипуляций и даже для шифрования данных RPC-сообщений. Понятно, что более высокие уровни аутентификации требуют более объемной обработки. Клиент может также дополнительно определить для сервера *имя участника* (principal name). Участником называется объект, распознаваемый системой безопасности RPC. Сервер должен зарегистрировать в SSP свое имя участника, отвечающего требованиям SSP.

SSP берет на себя все тонкости выполнения аутентификации сетевой коммуникации и шифрования не только для RPC, но также и для Winsock. Windows имеет несколько встроенных SSP-поставщиков, включая Kerberos SSP, для реализации аутентификации Kerberos версии 5 (включая поддержку AES), и защищенный канал Secure Channel (SChannel), реализующий протокол защищенных сокетов Secure Sockets Layer (SSL), и протокол безопасности транспортного уровня Transport Layer Security (TLS). SChannel также поддерживает в качестве надстройки над протоколами расширения TLS и SSL, позволяющие использовать шифр AES, а также шифры криптографии эллиптических кривых — elliptic curve cryptographic (ECC). Также, поскольку им поддерживается открытый криптографический интерфейс — open cryptographic interface (OCI) — и возможности гибкой криптографии — crypto-agile capabilities, — SChannel позволяет администратору заменять или добавлять существующие криптографические алгоритмы. В отсутствие специализированного SSP-провайдера программное обеспечение RPC использует встроенную защиту используемого транспорта. Встроенная защита имеется у нескольких транспортов, например у именованных каналов или у локальных RPC. Другие транспорты, такие как TCP, защиты не имеют, и в таком случае RPC при отсутствии указанного SSP совершает незащищенные вызовы.

ПРИМЕЧАНИЕ

Использование незашифрованных RPC может вызвать серьезные проблемы безопасности для вашей организации.

Еще одним свойством RPC-безопасности является возможность сервера заимствовать идентичность безопасности клиента с помощью функции RpcImpersonateClient. После завершения сервером выполнения операций с заимствованием от имени клиента он возвращается к своей собственной идентичности безопасности путем вызова функции RpcRevertToSelf или функции RpcRevertToSelfEx (см. главу 6).

Реализация RPC

Реализация RPC изображена на рис. 7.8, где показано, что основанное на использовании RPC приложение связывается с DLL-библиотекой RPC времени выполнения (%SystemRoot%\System32\Rpcrt4.dll). Эта библиотека предоставляет функции маршализации и демаршализации для использования функциональными заглушками RPC-приложения, а также функции для отправки и получения маршализированных данных. DLL-библиотека RPC времени выполнения включает поддержку процедур для обработки RPC по сети, а также форму RPC, называемую локальным RPC-вызовом. Локальный RPC может использоваться для связи между двумя процессами, находящимися на одной и той же машине, и DLL-библиотека RPC времени выполнения использует в качестве локального сетевого API-средства усовершенствованной системы вызова локальных процедур — advanced local procedure call (ALPC) в режиме ядра (см. главу 3). Когда RPC основан на нелокальных механизмах коммуникации, DLL-библиотека RPC времени выполнения использует API Winsock или именованного канала.

Подсистема RPC (RPCSS — %SystemRoot%\System32\Rpcss.dll) реализована, как служба Windows. Сама RPCSS является RPC-приложением, которое связывается с собственными экземплярами на других системах для выполнения поиска имени, регистрации и динамического отображения конечной точки. (Чтобы не загромождать рис. 7.8, на нем не показана RPCSS, связанная с DLL-библиотекой RPC времени выполнения.)

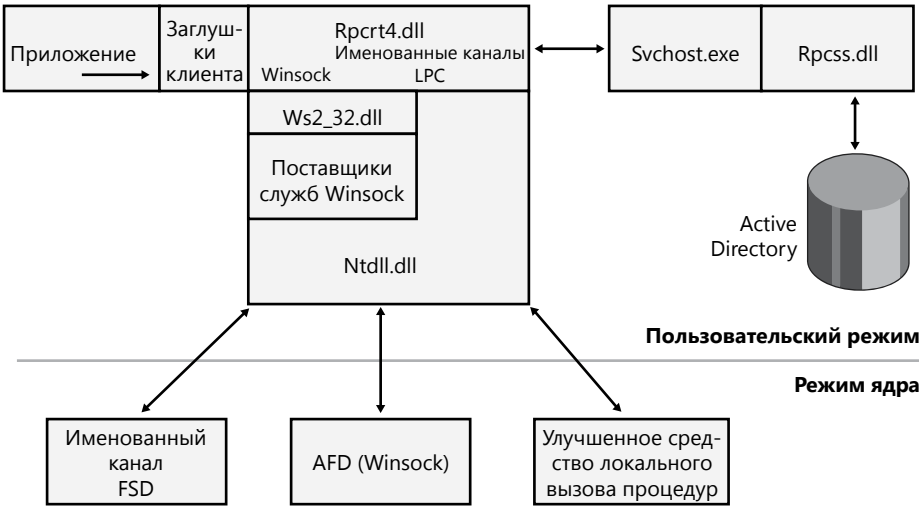


Рис. 7.8. Реализация RPC

В Windows также включена поддержка RPC в режиме ядра, осуществляемая с помощью RPC-драйвера режима ядра (%SystemRoot%\System32\Drivers\Msrpc.sys). RPC режима ядра предназначен для использования системой, и он реализован в качестве надстройки над ALPC. Winlogon включает RPC-сервер с документированным набором интерфейсов, к которому могут обращаться RPC-клиенты пользовательского режима, а Win32k.sys включает RPC-клиент, который связывается с Winlogon для внутренних уведомлений, например для об-

работки последовательности переноса внимания на безопасность работы — secure attention sequence (SAS). Имеющийся в Windows стек TCP/IP (а также WFP) также использует RPC режима ядра, чтобы связываться со службой интерфейса сетевого хранилища — Network Storage Interface (NSI), — которая обрабатывает информацию о конфигурации сети.

API-интерфейсы веб-доступа

Для облегчения разработки интернет-приложений Windows предоставляет как клиентские, так и серверные API-функции для работы с Интернетом. Используя эти API-интерфейсы, приложения могут предоставлять HTTP-службы и использовать FTP- и HTTP-службы, не вникая в сложности соответствующих протоколов. Клиентские API-интерфейсы включают компонент Windows Internet, также известный как WinInet, который позволяет приложениям взаимодействовать с протоколами FTP и HTTP, и компонент WinHTTP, который позволяет приложениям взаимодействовать с HTTP-протоколом. WinHTTP в определенных ситуациях более удобен, чем WinInet (в Windows-службах и в промежуточных приложениях). HTTP Server является API-интерфейсом серверной стороны, позволяющим вести разработку веб-серверных приложений.

WinInet

WinInet поддерживает протоколы HTTP, FTP Gopher. API-интерфейсы разбиты на подчиненные API-наборы, специфичные для каждого протокола. Используя API-функции, связанные с FTP, такую как InternetConnect для связи с HTTP-сервером, и далее используя функцию HttpOpenRequest, чтобы открыть дескриптор HTTP-запроса, функцию HttpSendRequestEx для отправки запроса серверу и получения ответа, функцию InternetWriteFile для отправки файла и функцию InternetReadFileEx для получения файла, разработчик приложения не вникает в детали установки соединения и форматирования TCP/IP-сообщений к различным протоколам. API-функции, связанные с HTTP, также предоставляют возможность постоянного использования cookie-файлов, кэширования файлов на стороне клиента и автоматическую обработку диалога, связанного с учетными данными. WinInet используется основными компонентами Windows, такими как Windows Explorer и Internet Explorer.

ПРИМЕЧАНИЕ

WinINet не поддерживает серверные реализации или использование службами. Для такого применения вместо него нужно использовать WinHTTP.

WinHTTP предоставляет абстракцию протокола HTTP v1.1 для клиентских приложений HTTP, аналогично тому, который предоставляется WinInet API-функциями, связанными с HTTP. Но если WinInet HTTP API-функции предназначены для интерактивных пользовательских приложений на стороне клиента, то WinHTTP API-функции разработаны для серверных приложений, связанных с HTTP-серверами. Серверные приложения зачастую реализуются как Windows-службы, которые не предоставляют пользовательского интерфейса, и поэтому не нуждаются в диалоговых окнах, отображаемых API-функциями WinInet. Кроме того, API-функции WinHTTP более приспособлены к увеличению масштабов

(например, поддерживают выгрузки на сервер файлов, размер которых превышает 4 Гбайт) и предлагают такие функциональные возможности обеспечения безопасности, как заимствование прав для выполнения потока, чего нельзя получить при использовании API-функций WinInet.

HTTP

Используя HTTP Server API, реализуемый Windows, серверные приложения могут регистрироваться для получения HTTP-запросов для конкретных URL-адресов, получать HTTP-запросы и отправлять HTTP-ответы. HTTP Server API включает поддержку SSL, благодаря чему приложения могут обмениваться данными по защищенным HTTP-соединениям. API включает возможности кэширования на серверной стороне, синхронные и асинхронные модели ввода-вывода, и как IPv4, так и IPv6 адресацию. Серверные API HTTP используются IIS и другими службами Windows, полагающимися на HTTP как на транспорт.

HTTP Server API, доступ к которому приложения получают через %SystemRoot%\System32\Httpapi.dll, зависит от драйвера режима ядра %SystemRoot%\System32\Drivers\Http.sys. Драйвер Http.sys запускается по первому же запросу любого приложения с помощью системного вызова HttpInitialize. Затем приложения вызывают функцию HttpCreateServerSession, чтобы инициализировать сеанс работы сервера для HTTP Server API. Затем они используют функцию HttpCreateRequestQueue для создания закрытой очереди запросов и функцию HttpCreateUrlGroup для создания URL-группы, определяющей URL-адреса, с которых им нужно обрабатывать запросы с помощью функции HttpAddUrlToUrlGroup. Используя очереди запросов и их зарегистрированные URL-адреса (которые они связывают с помощью функции HttpSetUrlGroupProperty), Http.sys позволяет более чем одному приложению обслуживать HTTP-запросы на заданном порту (например, на порту 80), чтобы каждое из них обслуживало HTTP-запросы к различным частям пространства имен URL-адресов, как показано на рис. 7.9.

Функция HttpReceiveHttpRequest получает входящие запросы, направленные на зарегистрированные URL-адреса, а функция HttpSendHttpResponse отправляет HTTP-ответы. Обе функции предлагают асинхронные операции для того, чтобы приложение могло использовать функцию GetOverlappedResult или порты завершения ввода-вывода, чтобы определить момент завершения операции.

Приложения могут использовать Http.sys для кэширования данных в невыгружаемой физической памяти путем вызова функции HttpAddFragmentToCache и связывания имени фрагмента (указанного как URL-префикс) с кэшированными данными. Http.sys вызывает функцию диспетчера памяти MmAllocatePagesForMdlEx для выделения неотображаемых физических страниц. (Для больших запросов Http.sys также пытается использовать большие страницы для оптимизации доступа к буферизированным данным.) Когда Http.sys требует отображение виртуального адреса на физическую память, которое описывается записью в кэше, например, когда он копирует данные в кэш или отправляет данные из кэша, он использует функцию MmMapLockedPagesSpecifyCache, а затем функцию MmUnmapLockedPages, после того как завершит свой доступ. Http.sys обслуживает кэшированные данные, пока приложение их не посчитает недей-

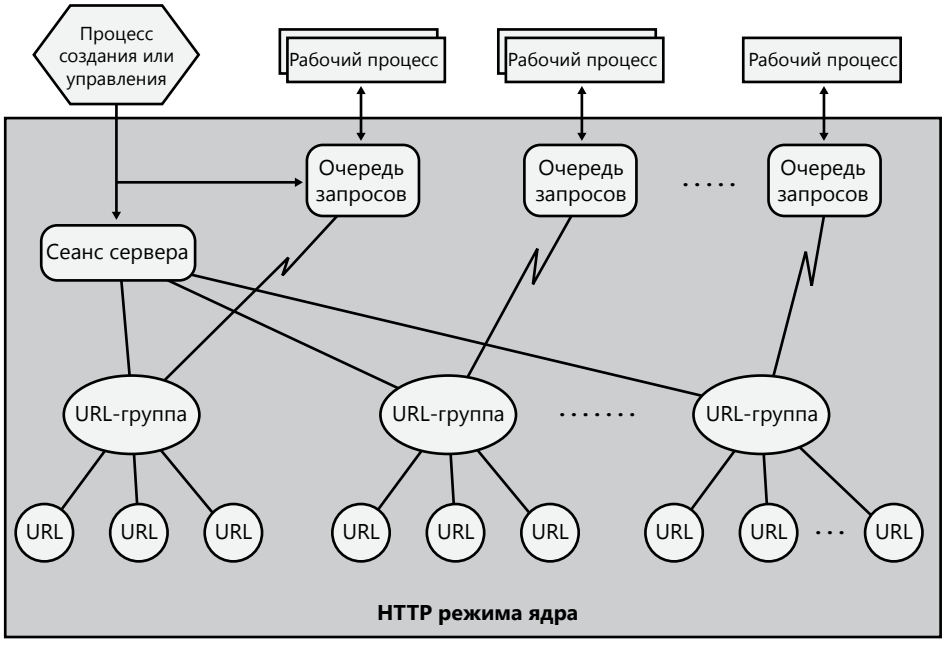


Рис. 7.9. Очереди HTTP-запросов и URL-группы

ствительными или пока не истечет дополнительно указанный приложением срок годности данных. Http.sys также урезает кэшированные данные в пробуждаемом рабочем потоке, когда поступает сигнал о событии уведомления о низком уровне памяти. Когда приложение указывает одно или несколько имен фрагментов в вызове функции HttpSendHttpResponse, драйвер Http.sys передает указатель на кэшированные данные в физической памяти драйверу TCP/IP, избегая тем самым операции копирования. Http.sys также содержит код для выполнения аутентификации на стороне сервера, включая полную поддержку SSL, благодаря чему исключается потребность обратного вызова к API пользовательского режима для выполнения шифрования и расшифровки трафика.

И наконец, HTTP Server API содержит множество настроек конфигурации, которые клиент может использовать для установки нужных параметров функционирования, например для настройки политик аутентификации, дросселирования пропускной способности, регистрации, ограничения количества подключений, состояния сервера, кэширования ответа привязки к SSL-сертификату.

Именованные каналы и почтовые слоты

Именованные каналы (named pipes) и почтовые слоты (mailslots) являются API-интерфейсами программирования для связи между процессами. Именованные каналы предоставляются для надежной двунаправленной связи, а почтовые слоты предоставляются для ненадежной, ненаправленной передачи данных. Преимуществом почтовых слотов является то, что они поддерживают возможность рассылки. В Windows оба API-интерфейса используют стандартные,

входящие в состав системы безопасности Windows-механизмы аутентификации и авторизации, позволяющие серверу четко контролировать подключение к ним конкретных клиентов.

Имена, назначаемые серверами именованным каналам и клиентам, соответствуют используемому в Windows универсальному соглашению об именовании — Universal Naming Convention (UNC), — являющемуся независимым от протоколов способом идентификации ресурсов в сети Windows. Реализация UNC-имен рассматривается в этой главе чуть позже.

Работа именованных каналов

Связь по именованным каналам состоит из сервера именованных каналов и клиента именованных каналов. Сервер именованных каналов является приложением, создающим именованный канал, к которому может подключиться клиент. Формат имени канала имеет вид *Сервер**Канал**Имя_канала*. Компонент имени *Сервер* указывает на компьютер, на котором выполняется программа сервера именованного канала. (Сервер именованного канала не может создать именованный канал на удаленной системе.) Имя может быть DNS-именем (например, mspress.microsoft.com), NetBIOS-именем (mspress) или IP-адресом (131.107.0.1). Компонент имени *Канал* должен быть строкой «Pipe», а компонент имени *Имя_канала* должен быть уникальным именем, присвоенным именованному каналу. Уникальная часть имени именованного канала может включать подкаталоги, примером имени именованного канала с подкаталогом может послужить \\MyComputer\Pipe\MyServerApp\ConnectionPipe.

Сервер именованного канала использует для создания такого канала Windows-функцию CreateNamedPipe. Одним из входных параметров этой функции является указатель на имя канала в формате \\.\Pipe*Имя_канала*. Группа символов «\\.\» является определенным в Windows псевдонимом, означающим «эта система», потому что канал должен быть создан на локальной системе (хотя доступ к нему может быть из удаленной системы). Другие параметры, принимаемые функцией, включают необязательный дескриптор безопасности, который защищает доступ к именованному каналу, флаг, указывающий, должен ли канал быть двунаправленным или однонаправленным, значение, показывающее максимальное количество поддерживаемых каналом одновременных подключений, и флаг, определяющий, в каком режиме должен работать канал, в *байтовом режиме* или в *режиме сообщений*.

Большинство сетевых API-интерфейсов работают только в байтовом режиме, а это значит, что сообщение, отправленное одной функцией отправки, может потребовать выполнения нескольких операций получения, составляющих полное сообщение из фрагментов. Именованный канал, работающий в режиме сообщений, упрощает реализацию получателя, поскольку между запросами на отправку и запросами на получение устанавливается соотношение один к одному. Поэтому получателю по завершении операции получения доставляется все сообщение целиком и ему не нужно заниматься отслеживанием фрагментов сообщения.

Первый вызов функции CreateNamedPipe с указанием конкретного имени приводит к созданию первого экземпляра под этим именем и определяет поведение всех экземпляров именованного канала, имеющих это имя. Сервер создает дополнительные экземпляры, вплоть до максимального количества, указанного

при первом вызове функции, с помощью дополнительных вызовов функции CreateNamedPipe. После создания хотя бы одного экземпляра именованного канала сервер выполняет Windows-функцию ConnectNamedPipe, которая позволяет именованному каналу, созданному сервером, установить соединение с клиентом. Функция ConnectNamedPipe может быть выполнена в синхронном или в асинхронном режиме и не завершает свою работу, пока клиент не установит соединение с экземпляром (или пока не будет выдана ошибка).

Для соединения с сервером клиент именованного канала использует Windows-функцию CreateFile или CallNamedPipe, указывая имя канала, созданного сервером. Если сервер выполнил вызов функции ConnectNamedPipe, профиль безопасности клиента и доступ, который им запрашивается к каналу (по чтению, по записи), проверяются на соответствие дескриптору безопасности канала (см. главу 6). Если клиенту предоставлен доступ к именованному каналу, он получает дескриптор, представляющий клиентскую сторону соединения по именованному каналу, и вызов сервером функции ConnectNamedPipe завершается.

После установки соединения по именному каналу клиент и сервер могут использовать для чтения из канала и для записи в него Windows-функции ReadFile и WriteFile. Для передачи сообщений именованные каналы поддерживают как синхронные, так и асинхронные операции, в зависимости от того, как был открыт дескриптор канала. На рис. 7.10 показана связь между сервером и клиентом по экземпляру именованного канала.

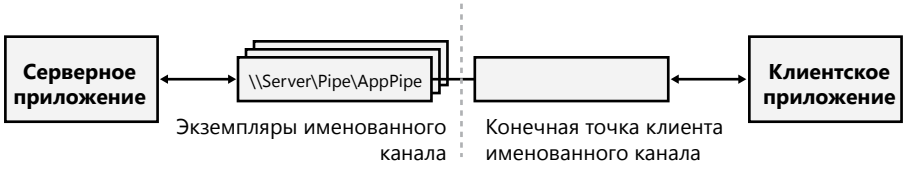


Рис. 7.10. Связь по именованному каналу

Сетевой API именованного канала характерен еще тем, что он позволяет серверу заимствовать права клиента путем использования функции ImpersonateNamedPipeClient[1]. Второй усовершенствованной областью функциональных возможностей API именованного канала является разрешение выполнять атомарные операции отправки и получения через API-функцию TransactNamedPipe. Эта функция ведет себя в соответствии с простой моделью транзакций, в которой сообщение и отправляется, и получается в рамках одной и той же операции. Иными словами, при выполнении этой функции в одной операции сочетаются операция записи и операция чтения за счет того, что операция записи не завершается до тех пор, пока получателем не будет завершена операция чтения.

Работа почтовых слотов

Почтовые слоты предоставляют ненадежный, однонаправленный, многоадресный сетевой транспорт. *Многоадресность* (multicast) является термином, использу-

[1] Вопрос использования заимствования прав в клиент-серверном приложении рассматривается в главе 6, в разделе «Заимствование прав».

емым для описания отправителя, посылающего в сеть сообщение одному или нескольким конкретным слушателям, что отличается от *широковещательной рассылки* (broadcast), которую будут получать все системы. Примером приложения, которое может использовать данный тип связи, может стать служба синхронизации времени, которая может отправлять время источника по всему домену каждые несколько секунд. Такое сообщение будут получать все приложения, слушающие конкретный почтовый слот. Получение сообщения о времени источника не критично для отдельно взятого компьютера сети (поскольку обновления времени отправляются относительно часто), поэтому сообщения о времени источника являются хорошим примером использования почтовых слотов, поскольку утрата сообщения обойдется без нанесения ущерба.

Аналогично именованным каналам, почтовые слоты интегрированы с Windows API. Сервер почтовых слотов создает почтовый слот с помощью функции CreateMailslot. Этой функции в качестве входного параметра передается UNC-имя в формате \\.\Mailslot*Имя_почтового_слота*. И опять по аналогии с именованными каналами, сервер почтовых слотов может создать почтовый слот только на той машине, на которой он выполняется, и имя, присваиваемое почтовому слоту, может включать подкаталоги. Функции CreateMailslot также передается дескриптор безопасности, управляющий пользовательским доступом к почтовому слоту. Дескрипторы, возвращаемые функцией CreateMailslot, *совмещаются*, что означает, это операции, выполняемые с этими дескрипторами, например отправка и получение сообщений, проводятся в асинхронном режиме.

Поскольку почтовые слоты являются однонаправленными и ненадежными, функции CreateMailslot не передается такое же большое количество параметров, как функции CreateNamedPipe. После того как эта функция создаст почтовый слот, сервер просто слушает поступающие клиентские сообщения, выполняя функцию ReadFile в отношении дескриптора, представляющего почтовый слот.

Клиенты почтового слота используют формат именования, подобный тому, который используется клиентами именованного канала, но с вариациями, позволяющими отправлять сообщения всем почтовым слотам с заданным именем в пределах клиентского домена или указанного домена. Чтобы отправить сообщение конкретному экземпляру почтового слота, клиент вызывает функцию CreateFile, указав конкретное имя компьютера. Примером такого имени может послужить имя *Сервер*\Mailslot*Имя_почтового_слота*. (Для представления локального компьютера клиент может указать символы «\\.\».) Если клиент хочет получить дескриптор, представляющий все почтовые слоты с заданным именем на домене, в состав которого он входит, он указывает имя в формате *\Mailslot*Имя_почтового_слота*, а если клиент хочет отправить сообщение всем почтовым слотам заданного имени в пределах другого домена, должен использовать следующий формат: *Имя_домена*\Mailslot*Имя_почтового_слота*.

После получения дескриптора, представляющего клиентскую сторону почтового слота, клиент отправляет сообщения путем вызова функции WriteFile. Из-за способа реализации почтовых слотов могут оправляться только сообщения, не превышающие по длине 424 байт. Если сообщение больше 424 байт, реализация почтового слота использует надежный механизм связи, требующий клиент-серверного соединения «один к одному», что препятствует реализации многоадресной отправки. Это ограничение делают почтовые слоты абсолютно

бесполезными для передачи сообщений, превышающих по размеру 424 байта. На рис. 7.11 показан пример клиента, осуществляющего широковещательную рассылку сразу нескольким серверам почтовых слотов внутри домена.

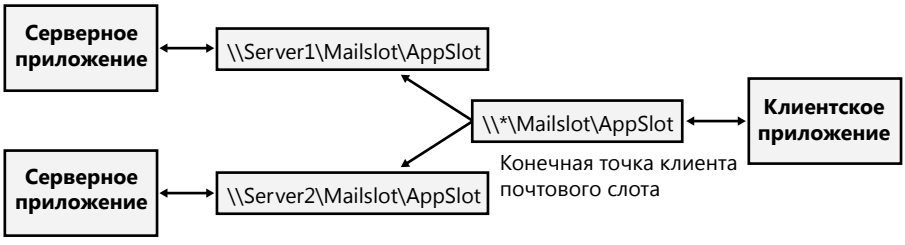


Рис. 7.11. Широковещательная рассылка, осуществляемая по почтовым слотам

Реализация именованных каналов и почтовых слотов

В качестве доказательства своей тесной интеграции с Windows, все функции именованных каналов и почтовых слотов реализованы в библиотеке Kernel32.dll, которая является Windows DLL-библиотекой на стороне клиента. Функции ReadFile и WriteFile, являющиеся функциями, которые используются приложениями для отправки и получения сообщений с использованием именованных каналов и почтовых слотов, являются в Windows основными процедурами ввода-вывода. Функция CreateFile, которую клиент использует для открытия либо именованного канала, либо почтового слота, также является в Windows стандартной процедурой ввода-вывода. Но имена, указываемые приложениями именованных каналов и почтовых слотов, как показано на рис. 7.12, указывают пространство имен файловой системы, управляемое драйвером файловой системы именованных каналов (%SystemRoot%\System32\Drivers\Npfs.sys) и драйвером файловой системы почтовых слотов (%SystemRoot%\System32\Drivers\Msfs.sys).

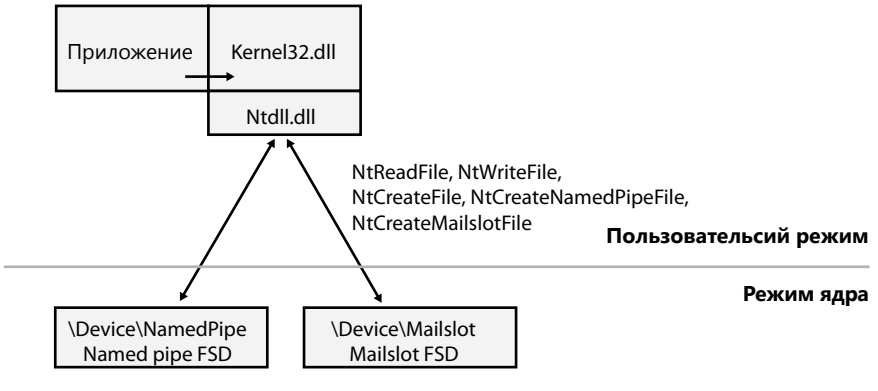


Рис. 7.12. Реализация именованного канала и почтового слота

Драйвер файловой системы именованных каналов создает объект устройства \Device\NamedPipe и символическую ссылку на этот объект \Global??\Pipe. Драйвер файловой системы почтовых слотов создает объект устройства

\Device\Mailslot и символическую ссылку на этот объект \Global??\Mailslot, которая указывает на этот объект устройства. (Объяснения, касающиеся каталога диспетчера объектов \Global??, даны в главе 3.) Имена, передаваемые функции CreateFile в формате \\.\Pipe\... и \\.\Mailslot\..., имеют префикс «\\.\», который переводится в «\Global??\», так что имена разрешаются посредством символической ссылки на объект устройства. Специальные функции CreateNamedPipe и CreateMailslot используют соответствующие исходные функции NtCreateNamedPipeFile и NtCreateMailslotFile, которые в конечном счете вызывают функцию IoCreateFile.

Чуть позже в данной главе будет рассмотрено, как задействуется драйвер перенаправления файловой системы, когда имя, указанное удаленным именованным каналом или почтовым слотом, разрешается в удаленной системе Но, когда именованный канал или почтовый слот создаются сервером или открываются клиентом, в конечном счете вызывается соответствующий драйвер файловой системы — file-system driver (FSD) — той машины, где находится именованный канал или почтовый слот. Причина реализации именованных каналов и почтовых слотов на FSD-драйверах объясняется тем, что они могут воспользоваться существующей инфраструктурой в диспетчере объектов, диспетчером ввода-вывода, редиректором (рассматриваемым далее в этой главе) и протоколом блока сообщений сервера — Server Message Block (SMB). Эта интеграция приводит к ряду преимуществ:

- Для реализации стандартной безопасности Windows для именованных каналов и почтовых слотах FSD-драйверы используют функции безопасности режима ядра.
- Поскольку FSD-драйверы интегрированы с пространством имен диспетчера объектов для открытия именованного канала или почтового слота, приложения могут использовать функцию CreateFile.
- Для взаимодействия с именованными каналами и почтовыми слотами приложения могут использовать такие функции Windows, как ReadFile и WriteFile.
- Для отслеживания счетчиков дескрипторов и ссылок для файловых объектов представляющие именованные каналы и почтовые слоты FSD-драйверы полагаются на диспетчер объектов.
- FSD-драйверы могут реализовывать свои собственные пространства имен именованных каналов и почтовых слотов, дополненные подкаталогами.

ЭКСПЕРИМЕНТ: ВЫВОД СПИСКА ПРОСТРАНСТВ ИМЕН ИМЕНОВАННЫХ КАНАЛОВ И НАБЛЮДЕНИЕ ЗА АКТИВНОСТЬЮ ЭТИХ КАНАЛОВ

Использовать Windows API для открытия корневого каталога FSD именованного канала и вывода списка каталогов невозможно, но это можно сделать с помощью исходных API-служб. Имена именованных каналов, определенных на компьютере, а также количество экземпляров, созданных для имени, и максимальное количество экземпляров, определенное в ходе вызова сервером функции CreateNamedPipe, можно увидеть с помощью утилиты PipeList из набора Sysinternals. Пример вывода утилиты PipeList имеет следующий вид:

```
C:\>pipelist
PipeList v1.01
```

```
by Mark Russinovich
http://www.sysinternals.com
Pipe Name                                  Instances       Max Instances
---------                                  ---------       -------------
InitShutdown                                   3               -1
lsass                                          6               -1
protected_storage                              3               -1
ntsvcs                                         3               -1
scerpc                                         3               -1
net\NtControlPipe1                             1                1
plugplay                                       3               -1
net\NtControlPipe2                             1                1
Winsock2\CatalogChangeListener-394-0           1                1
epmapper                                       3               -1
Winsock2\CatalogChangeListener-25c-0           1                1
LSM_API_service                                3               -1
net\NtControlPipe3                             1                1
eventlog                                       3               -1
net\NtControlPipe4                             1                1
Winsock2\CatalogChangeListener-3f8-0           1                1
net\NtControlPipe5                             1                1
net\NtControlPipe6                             1                1
net\NtControlPipe0                             1                1
atsvc                                          3               -1
Winsock2\CatalogChangeListener-438-0           1                1
Winsock2\CatalogChangeListener-2c8-0           1                1
net\NtControlPipe7                             1                1
net\NtControlPipe8                             1                1
net\NtControlPipe9                             1                1
net\NtControlPipe10                            1                1
net\NtControlPipe11                            1                1
net\NtControlPipe12                            1                1
142CDF96-10CC-483c-A516-3E9057526912           1                1
net\NtControlPipe13                            1                1
net\NtControlPipe14                            1                1
TSVNCache-000000000001b017                    20               -1
TSVNCacheCommand-000000000001b017              2               -1
Winsock2\CatalogChangeListener-2b0-0           1                1
Winsock2\CatalogChangeListener-468-0           1                1
TermSrv_API_service                            3               -1
Ctx_WinStation_API_service                     3               -1
PIPE_EVENTROOT\CIMV2SCM EVENT PROVIDER         2               -1
net\NtControlPipe15                            1                1
keysvc                                         3               -1
```

При изучении данного вывода становится понятно, что именованные каналы используются в качестве механизма связи несколькими системными компонентами. Например, канал InitShutdown создан WinInit для получения удаленных команд на завершение работы, а канал Atsvc создан службой

Планировщик заданий, чтобы позволить приложениям связываться с ней для планирования заданий. Используя средство поиска объектов утилиты Process Explorer, вы можете определить, какой процесс имеет открытыми каждый из этих каналов. ■

ПРИМЕЧАНИЕ

Значение максимального количества экземпляров — Max Instances, — равное –1, означает, что верхнее ограничение для количества экземпляров не установлено.

NetBIOS

До 1990-х годов API-интерфейс программирования основной системы сетевого ввода-вывода — Network Basic Input/Output System (NetBIOS) — был наиболее широко используемым API сетевого программирования на PC-компьютерах. NetBIOS позволяет создавать как надежные, ориентированные на установку соединения, так и ненадежные, не ориентированные на установку соединения обмена данными. Windows поддерживает NetBIOS для своих устаревших приложений. Microsoft не рекомендует разработчикам приложений использовать NetBIOS, поскольку другие API-интерфейсы, такие как именованные каналы и Winsock, обладают намного большей гибкостью и переносимостью. NetBIOS поддерживается в Windows протоколом TCP/IP.

Имена NetBIOS

NetBIOS полагается на соглашение об именовании, в соответствии с которым компьютерам и сетевым службам даются 16-байтные NetBIOS-имена. Шестнадцать байт имени NetBIOS рассматриваются как модификатор, который может указать на уникальность имени или на его принадлежность к группе. Сети может быть назначен только один уникальный экземпляр имени NetBIOS, но несколько приложений могут назначить одно и то же имя группы. Клиент может оправлять многоадресные сообщения путем отправки их на имя группы.

Для поддержки функциональной совместимости с системами Windows NT 4, а также с системами Windows 9x/Me Windows автоматически определяет имя NetBIOS для домена, которое включает до 15 первых байтов из левой части имени системы доменных имен — Domain Name System (DNS), — которое администратор назначает домену. Например, если домен был назван mspress.microsoft.com, NetBIOS-имя домена будет mspress.

Другим понятием, используемым NetBIOS, являются номера LAN-адаптера (LANA). LANA-номер назначается каждому NetBIOS-совместимому протоколу, находящемуся уровнем выше сетевого адаптера. Например, если у компьютера есть два сетевых адаптера и TCP/IP и NWLink могут использовать любой из них, будет назначено четыре LANA-номера. Важность LANA-номеров обусловливается тем, что NetBIOS-приложения должны явным образом назначать свое служебное имя каждому LANA, через который оно собирается допускать клиентские подключения. Если приложение прислушивается к клиентским подключениям к конкретному имени, клиенты могут получить доступ к имени только через протоколы сетевых адаптеров, для которых это имя зарегистрировано.

Работа NetBIOS

Серверные приложения NetBIOS используют NetBIOS API для нумерации LANA-адаптеров, присутствующих в системе, и назначения NetBIOS-имен, представляющих службу приложения для каждого LANA. Если сервер является ориентированным на установку соединения, он выполняет NetBIOS-команду listen для ожидания попыток подключения со стороны клиента. После того как клиент подключится, сервер выполняет NetBIOS-функции для отправки и получения данных. Связь без установки соединения работает аналогичным образом, но сервер просто читает сообщения без установки соединений.

Клиенты, ориентированные на установку соединения, используют NetBIOS-функции для установки соединения с сервером NetBIOS, а затем выполняют дополнительные NetBIOS-функции для отправки и получения данных. Установленное NetBIOS-соединение известно также как сеанс. Если клиент хочет отправить сообщение без установки соединения, он просто указывает функции send NetBIOS-имя сервера.

NetBIOS состоит из множества функций, но все они проходят через один и тот же интерфейс Netbios. Такая схема процедур явилась результатом преемственности из тех времен, когда система NetBIOS была реализована на MS-DOS в качестве службы прерываний MS-DOS. NetBIOS-приложение будет выполнять MS-DOS-прерывание и передавать структуру данных реализации NetBIOS, в которой определен каждый аспект выполняемой команды. В результате Netbios-функция в Windows получает всего один параметр, являющийся структурой данных, которая содержит параметры, характерные для службы, запрашиваемой приложением.

ЭКСПЕРИМЕНТ: ИСПОЛЬЗОВАНИЕ NBTSTAT ДЛЯ ПРОСМОТРА NETBIOS-ИМЕН

Для вывода списка имеющихся в системе активных сеансов, отображения имен NetBIOS на TCP/IP, кэшированного на компьютере и определенных на компьютере NetBIOS-имен, можно воспользоваться включенной в Windows командой Nbtstat. Ниже представлен пример выполнения команды Nbtstat с ключом –n, которая выводит список NetBIOS-имен, определенных на компьютере:

```
C:\Users\Toby>nbtstat -n
Local Area Connection:
Node IpAddress: [192.168.0.193] Scope Id: []
                NetBIOS Local Name Table
       Name               Type         Status
    ---------------------------------------------
    WIN-NLRTEOW2ILZ<00>  UNIQUE      Registered
    WORKGROUP      <00>  GROUP       Registered
    WIN-NLRTEOW2ILZ<20>  UNIQUE      Registered
```

■

Реализация NetBIOS API

Компоненты, с помощью которых реализован интерфейс NetBIOS API, показаны на рис. 7.13. Функция Netbios экспортируется приложениям библиотекой %SystemRoot%\System32\Netbios.dll. Библиотека Netbios.dll открывает дескриптор драйвера режима ядра, называемый *NetBIOS-эмулятором* (%SystemRoot%\System32\Drivers\Netbios.sys), и от имени приложения выдает файловые Windows-команды DeviceIoControl. NetBIOS-эмулятор переводит NetBIOS-

команды, выдаваемые приложением, в TDI-команды, отправляемые драйверам протоколов.

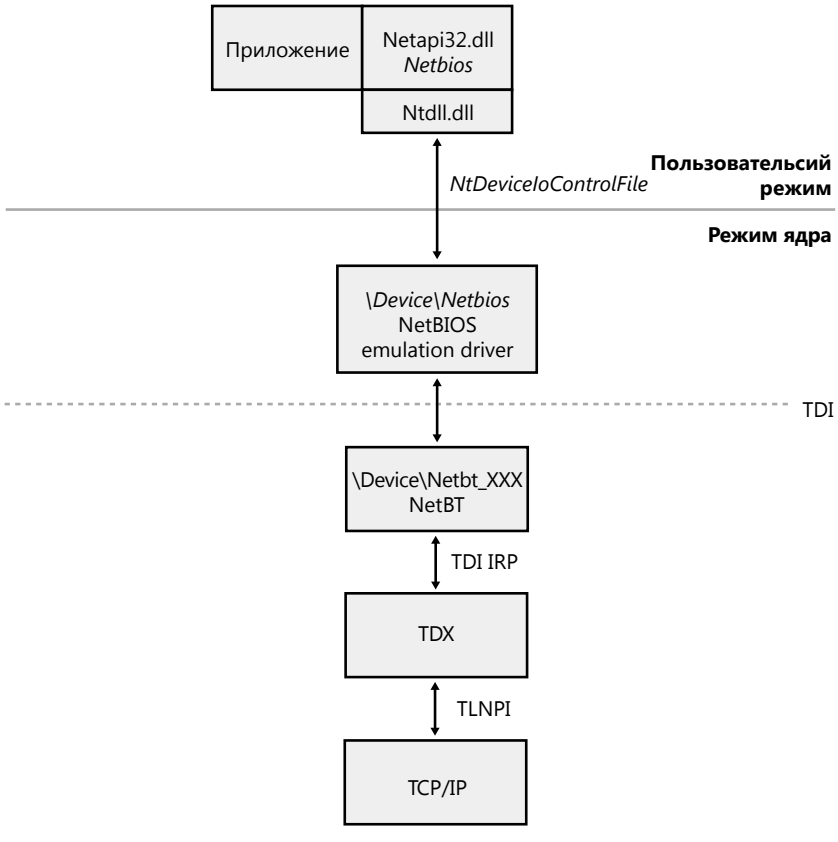


Рис. 7.13. Реализация NetBIOS API

Если приложению нужно использовать NetBIOS поверх протокола TCP/IP, NetBIOS-эмулятор требует присутствия NetBT-драйвера (%SystemRoot%\System32\Drivers\Netbt.sys). Драйвер NetBT известен как драйвер NetBIOS, надстроенный над TCP/IP, и он отвечает за поддержку семантики NetBIOS, которая присуща расширенному протоколу пользовательского интерфейса NetBIOS — NetBIOS Extended User Interface (NetBEUI), включенному в предыдущие версии Windows, но не протоколу TCP/IP. Например, NetBIOS полагается на имеющуюся в NetBEUI передачу в режиме сообщений и на средство NetBIOS по разрешению имен, поэтому драйвер NetBT реализует их поверх протокола TCP/IP.

Другие сетевые API

Windows включает другие сетевые API, которые используются намного реже или представляют собой уровни уже рассмотренных API-интерфейсов (которые выходят за рамки этой книги). И тем не менее, пять из них:

- фоновая интеллектуальная служба передачи — Background Intelligent Transfer Service (BITS),

- распределенная модель компонентных объектов — Distributed Component Object Model (DCOM);
- служба очереди сообщений — Message Queuing (MSMQ);
- одноранговая инфраструктура — Peer-to-Peer Infrastructure (P2P)
- и универсальный стандарт Plug and Play — Universal Plug and Play (UPnP) с расширениями Plug and Play — Plug and Play Extensions (PnP-X),

настолько важны для работы Windows-системы и многих приложений, что заслуживают кратких описаний.

Фоновая интеллектуальная служба передачи

Фоновая интеллектуальная служба передачи — Background Intelligent Transfer Service (BITS) — является как службой, так и API-интерфейсом, предоставляющим надежную асинхронную передачу файлов между системами с использованием либо SMB, HTTP, либо HTTPS-протокола. Обычно BITS работает в фоновом режиме, пользуясь незанятой пропускной возможностью сети путем отслеживания интенсивности использования сети, и дросселирует саму себя, потребляя только невостребованные ресурсы, но BITS-передачи могут также вестись и на первом плане и состязаться за ресурсы с другими процессами, запущенными в системе.

BITS отслеживает ситуацию или работает по запланированному сценарию передач, то есть выполняет то, что известно как задания передачи для каждого пользователя (не нужно путать их с заданиями и с объектами заданий, рассмотренными в главе 5). Каждое задание является записью в очереди и описывает передаваемый файл, контекст безопасности (маркер доступа) под которым нужно работать, и приоритет задания. BITS версии 4.0 интегрирована в механизм BranchCache (рассматриваемый далее в этой главе) для дальнейшего сокращения занимаемой полосы пропускания сети.

BITS используется многими другими компонентами Windows, например Microsoft Update, Windows Update, Internet Explorer (версии 9 и выше, для загрузки файлов), Microsoft Outlook (для загрузки адресных книг), Microsoft Security Essentials (для загрузки ежедневных обновлений вирусных сигнатур) и другими, что делает BITS наиболее широко используемой на сегодняшний день сетевой системой передачи файлов.

BITS предоставляет следующие возможности:

- **Цельную передачу данных.** Компоненты создают задания передачи BITS, которые затем будут выполняться, пока будут передаваться файлы. Когда пользователь выходит из системы, система перезапускается или у системы пропадает подключение к сети, BITS приостанавливает передачу. Передача возобновляется с того места, на котором она остановилась, как только пользователь снова войдет в систему или будет восстановлено подключение к сети. Приложению, создающему задание передачи, не нужно оставаться в системе, пока будет вестись передача. Задания передачи, созданные под учетной записью службы (например, Windows Update), всегда считаются остающимися в системе, позволяя этим заданиям выполняться непрерывно.
- **Несколько типов передачи.** BITS поддерживает три типа передачи: загрузка (от сервера к клиенту), выгрузка (от клиента к серверу) и выгрузка с ответом (от клиента к серверу с уведомлением, получаемым от сервера).

- **Назначение приоритетов передач.** Когда создается задача передачи, ей указывается приоритет (переднего плана — Foreground, высокий фоновый — Background High, обычный фоновый — Background Normal, или низкий фоновый — Background Low). Все задания фонового приоритета используются только на невостребованных сетевых ресурсах, а задания с приоритетом переднего плана состязаются с приложениями за сетевые ресурсы. Если есть несколько заданий, BITS обрабатывает их в порядке приоритетности, используя циклическую систему планирования в пределах конкретного приоритета, поэтому все задания получают возможность прогрессировать в передаче данных.
- **Безопасная передача данных.** BITS обычно выполняет задание передачи, используя контекст безопасности создания задачи, но вы можете также использовать BITS API для указания учетных данных для заимствования прав пользователя. Для обеспечения секретности при работе в сети нужно воспользоваться протоколом HTTPS.
- **Управление.** BITS API состоит из методов для создания, запуска, остановки, отслеживания, подсчета, модификации или запроса уведомления об изменении состояния задания передачи. В состав инструментария входит утилита BITSAdmin, использовать которую не рекомендуется и которая будет удалена из будущих версий Windows, и командлеты Windows PowerShell, являющиеся более предпочтительным механизмом управления.

При загрузке файлов BITS записывает файл во временный скрытый файл в каталоге назначения. Разумеется, BITS будет заимствовать права пользователя, чтобы обеспечить безопасность файловой системы и правильное принудительное применение квот. Когда приложение вызывает метод IBackgroundCopyJob::Complete (или командлет Complete-BitsTransfer в PowerShell), BITS переименовывает временные файлы, давая им предназначенные имена, и файлы становятся доступны клиенту. Если в каталоге назначения уже есть файл с таким же именем, BITS перезаписывает файл.

По умолчанию при выгрузке файлов BITS не разрешает перезапись существующего файла. Когда передача завершится и BITS станет перезаписывать файл, клиенту будет возвращена ошибка. Чтобы включить перезапись, нужно с помощью сценария PowerShell или сценария Windows Management Instrumentation (WMI) установить в метабазе Internet Information Services (IIS) для свойства BITSAllowOverwrites значение True.

Сервер BITS является компонентом серверной стороны, позволяющим настраивать IIS-сервер на разрешение BITS-клиентам выполнять передачу файлов в виртуальные каталоги IIS. После завершения выгрузки файла BITS-сервер может уведомить веб-приложение о присутствии нового файла (через POST-сообщение HTTP), чтобы веб-приложение могло обработать выгруженные на сервер файлы.

BITS-сервер расширяет IIS для поддержки дросселированной, перезапускаемой выгрузки файлов. Чтобы воспользоваться возможностью выгрузки, нужно создать на сервере виртуальный каталог IIS для выгрузки клиентом его файлов. BITS добавляет свойства к метабазе IIS для создаваемого вами виртуального каталога и использует эти свойства для определения того, как выгружать эти файлы.

Из соображений безопасности BITS не позволит выгружать в виртуальный каталог файлы с включенными разрешениями на выполнение сценариев и обычное выполнение. Если файл выгружается в виртуальный каталог, у которого включены эти разрешения, задание потерпит неудачу. Кроме того, BITS не требует, чтобы виртуальный каталог был с включенным доступом по записи, поэтому рекомендуется отключить для виртуального каталога доступ по записи, но у пользователя должен быть доступ по записи к физическому каталогу.

В некоторых случаях вместо IIS может использоваться BITS Compact Server. Compact Server предназначен для использования потребителями, имеющими отношение к производству и к малому бизнесу и отвечающими следующим условиям:

- Ожидаемое использование составляет максимум 25 групп URL-адресов, и каждая URL-группа поддерживает до трех одновременных передач файлов.
- Передачи файлов происходят между системами одного и того же домена или между доменами, пользующимися взаимным доверием.
- Передачи файлов не предназначены клиентам с выходом в Интернет.

На рис.7.14 показывается, как загрузить BITS-модуль в PowerShell, а также некоторые командлеты BITS PowerShell.

На рис. 7.15 показан пример работы утилиты BITSAdmin, использовать которую теперь не рекомендуется, отдавая предпочтение при управлении и использовании BITS оболочке PowerShell.

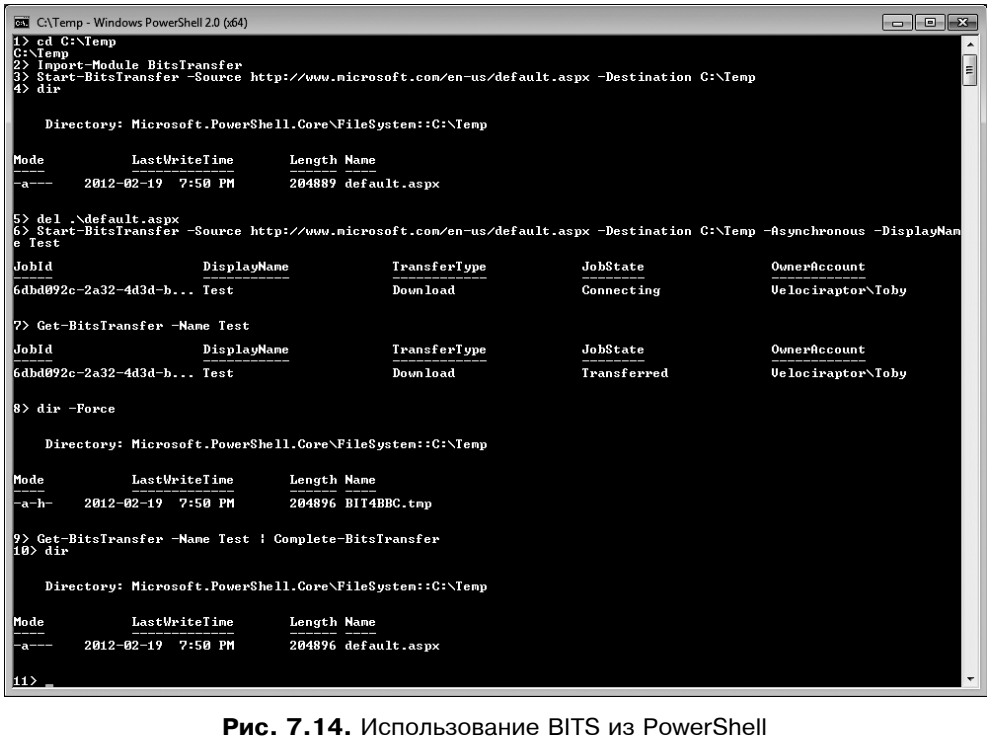


Рис. 7.14. Использование BITS из PowerShell

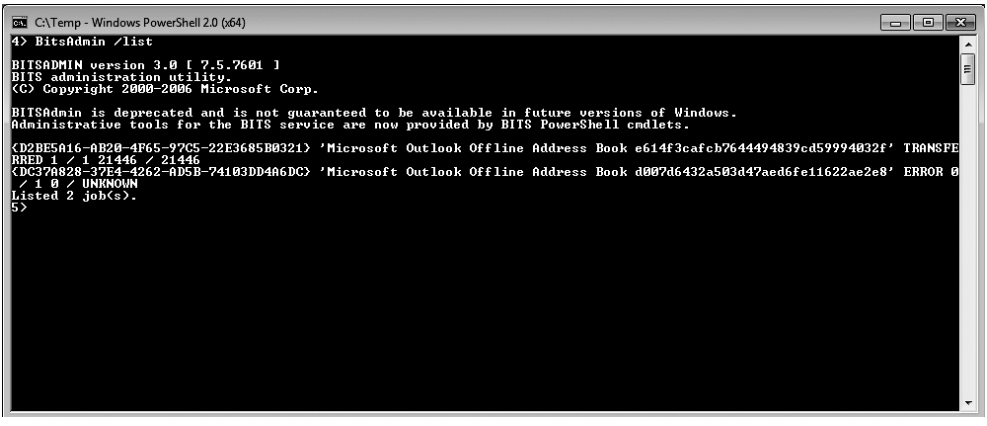


Рис. 7.15. Утилита BitsAdmin

На рис. 7.16 показаны сообщения BITS, записанные в журнал событий.

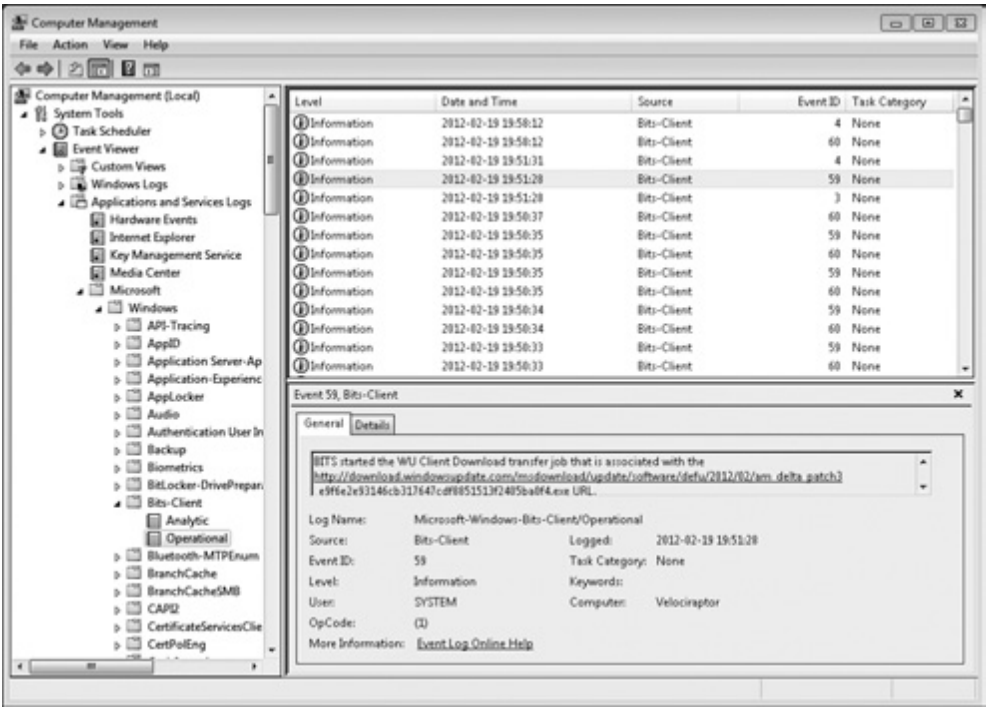


Рис. 7.16. Сообщения BITS в журнале событий

Одноранговая инфраструктура

Одноранговая или пиринговая инфраструктура (Peer-to-Peer Infrastructure) является набором API-интерфейсов, охватывающих различные технологии для усовершенствования сетевого стека Windows путем предоставления гибкой

поддержки одноранговой сети (peer-to-peer, P2P) для приложений и служб. P2P-инфраструктура охватывает четыре основные технологии, показанные на рис. 7.17.

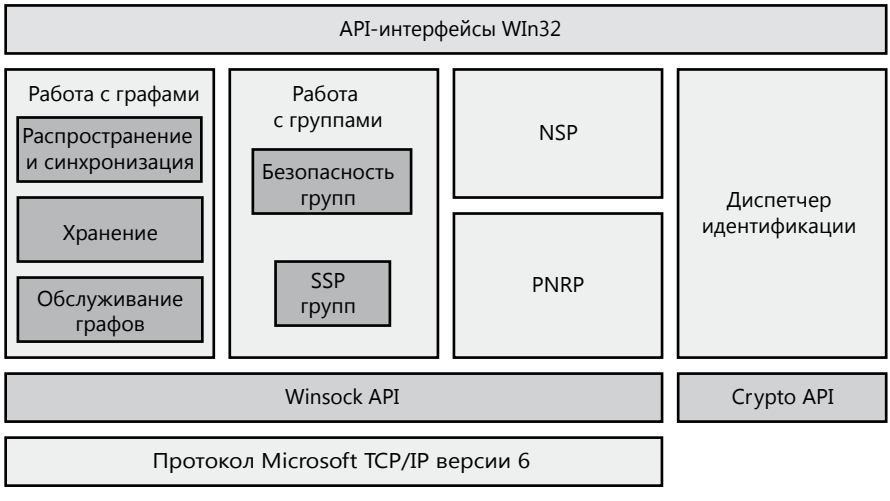


Рис. 7.17. Одноранговая архитектура

Основными компонентами одноранговой архитектуры являются следующие:

- **Механизм создания одноранговых графов — Peer-to-Peer Graphing.** Позволяет приложениям эффективно и надежно передавать данные между одноранговыми узлами сети путем использования узлов и событий.
- **Поставщик пространства имен одноранговой сети — Peer-to-Peer Namespace Provider.** Позволяет осуществлять разрешение имен узлов одноранговой сети (пиров) и их служб без использования сервера.
- **Механизм группировки одноранговой сети — Peer-to-Peer Grouping.** Объединяет технологии создания графов и пространства имен для группировки и изоляции служб и (или) пиров в определенной группе и для уникальной идентификации этой группы.
- **Диспетчер идентификации одноранговой сети — Peer-to-Peer Identity Manager.** Улучшает службы, предоставляемые поставщиком пространства имен для безопасного создания, публикации и идентификации имен пиров, а также для идентификации участников групп, являющихся частью API механизма группировки.

Имеющаяся в Windows инфраструктура одноранговой сети также составляет пару с совместным интерфейсом одноранговой сети — Peer-to-Peer Collaboration Interface, который добавляет поддержку создания совместных P2P-приложений (например, онлайновых игр и отправки групповых мгновенных сообщений) и вытесняет архитектуру связи реального времени — Real-Time Communications (RTC), — появившуюся в ранних версиях Windows. Эта инфраструктура также предусматривает наличие возможностей архитектуры People Near Me (PNM).

DCOM

Разработанный Microsoft COM API позволяет приложениям состоять из разных компонентов, где каждый компонент является заменяемым, самодостаточным модулем. COM-объект экспортирует объектно-ориентированный интерфейс к методам для работы с данными внутри объекта. Поскольку COM-объекты предоставляют четко определенные интерфейсы, разработчики могут реализовать новые объекты для расширения интерфейсов и динамически обновлять приложения новой поддержкой.

DCOM (Distributed Component Object Model — распределенная модель компонентных объектов) расширяет COM, позволяя компонентам приложения находиться на разных компьютерах, а это означает, что приложениям не нужно беспокоиться о том, что один COM-объект может быть на локальном компьютере, а другой может быть где-то в сети. Таким образом, DCOM предоставляет открытость размещения, упрощающую разработку распределенных приложений. DCOM не является самодостаточным API, поскольку при выполнении своей работы зависит от RPC.

Служба очереди сообщений

Служба очереди сообщений — Message Queuing — является платформой общего назначения для разработки распределенных приложений, использующих слабосвязанный обмен сообщениями (loosely coupled messaging). Поэтому служба очереди сообщений представляет собой API и инфраструктуру отправки сообщений. Источником ее гибкости является тот факт, что ее очереди служат хранилищами сообщений, в которых отправители могут выставлять очередь сообщений для получателей, а получатели могут по своему усмотрению извлекать сообщения из очереди. Отправители и получатели не должны устанавливать соединения, чтобы использовать службу очереди сообщений, и не нуждаются в одновременном выполнении, что позволяет организовать асинхронный обмен сообщениями без установки соединения.

Примечательным свойством службы очереди сообщений является то, что она интегрирована с сервером транзакций Microsoft Transaction Server (MTS) и SQL Server, поэтому она может участвовать в распределенных транзакциях Microsoft Distributed Transaction Coordinator (MS DTC). Использование MS DTC со службой очереди сообщений позволяет разрабатывать надежные функции транзакций для трехуровневых приложений.

UPnP с PnP-X

Универсальный стандарт Plug and Play является архитектурой для однорангового подключения к сети саморегулируемых приборов, устройств и *контрольных точек*. Он разработан с целью предоставления простого в использовании, гибкого, основанного на стандартах подключения к специализированным, управляемым или неуправляемым сетям, где сети могут быть домашними, сетями малого бизнеса или сетями, подключенными непосредственно к Интернету. Универсальный стандарт Plug and Play является распределенной, открытой сетевой архитектурой, использующей существующие TCP/IP и веб-технологии, чтобы дать возможность получения гладкой родственной сети вдобавок к управлению и передаче данных между сетевыми устройствами.

Универсальный стандарт Plug and Play поддерживает нулевую конфигурацию, незаметную работу сети и автоматическое обнаружение широкого спектра категорий устройств от многих производителей. Это позволяет устройству динамически подключаться к сети, получать IP-адрес, передать его возможности по запросу. Затем другие контрольные точки могут использовать Control Point API с технологией UPnP technology для изучения присутствия и возможностей других устройств. Устройство может беспрепятственно и автоматически покинуть сеть, когда оно больше не используется.

Расширения Plug and Play (PnP-X), показанные на рис. 7.18, являются дополнительным компонентом Windows, позволяющим подключенным к сети устройствам интегрироваться в ядре с диспетчером устройств Plug and Play. Используя PnP-X, подключенные к сети устройства отображаются в диспетчере устройств подобно устройствам, подключенным к локальной системе, и ведут себя при установке и управлении так же, как и локальное устройство. (Например, установка выполняется через стандартный мастер добавления нового оборудования.)

Рис. 7.18. Реализация PnP-X

Для обнаружения устройств, совместимых с PnP-X, в число которых входят UPnP-устройства (через Simple Service Discovery Protocol), и более новых устройств с профилем устройства для веб-служб (через протокол WS-Discovery) — Device Profile for Web Services (DPWS), PnP-X использует драйвер виртуальной сетевой шины, который использует службу перебора подключений к IP шине (%SystemRoot%\System32\Ipbusenum.dll). Служба перебора подключений к IP шине сообщает об обнаруженных устройствах диспетчеру устройств Plug and Play, который использует при необходимости службы диспетчера устройств Plug and Play пользовательского режима (например, для установки драйвера). Это похоже на беспроводное обнаружение (как в Bluetooth) и не похоже на обнаружение проводного устройства (как в USB), но PnP-X-перебор и установка драйвера должны быть запрошены явным образом пользователем из обозревателя сети Network Explorer.

ПРИМЕЧАНИЕ

DPWS v1.1 стал OASIS-стандартом в июне 2009, и его цели были сходны с целями UPnP, но он тесно интегрирован со стандартами и средами веб-служб и допускает большую степень расширяемости по сравнению с UPnP.

Поддержка нескольких редиректоров

Доступ к ресурсам файловых систем на удаленных машинах (которые часто называются совместным использованием файлов) приложения получают с помощью UNC-путей, например *имя_сервера**имя_общего_ресурса**файл*. Доступ к ресурсам может быть получен непосредственно с использованием UNC-имени, если оно уже известно, и учетные данные вошедшего в систему пользователя вполне достаточны. Кроме этого для определения общего количества компьютеров и ресурсов, которые эти компьютеры экспортируют для совместного использования, отображения буков дисков на UNC-пути и явного указания учетных данных может быть использован сетевой API-интерфейс Windows — Windows Networking (WNet) API. Для доступа к SMB-серверам со стороны клиента Microsoft поставляет SMB-клиента, у которого есть компонент режима ядра, называемый *мини-редиректором*, и компонент пользовательского режима, называемый *службой рабочей станции*. Microsoft также предоставляет доступ к редиректорам, которые могут обращаться к WebDAV-ресурсам, к ресурсам NFS v2/v3 (только в редакциях Профессиональная и Корпоративная) и к общим накопителям служб терминалов. Сторонние разработчики могут добавлять в Windows свои собственные редиректоры. В этом разделе будет рассмотрено программное обеспечение, принимающее решение, какой из редиректоров нужно задействовать для файлового доступа с использованием UNC-путей. За это отвечают следующие компоненты:

- *Маршрутизатор многосетевого доступа — Multiple Provider Router (MPR)*, являющийся DLL-библиотекой (%SystemRoot%\System32\Mpr.dll), которая определяет, к какой сети обращаться, когда приложение использует Windows WNet API для обзора удаленных файловых ресурсов.
- *Многосетевой UNC-поставщик — Multiple UNC Provider (MUP)*, являющийся драйвером (%SystemRoot%\System32\Drivers\Mup.sys), который определяет, к какой сети обращаться, когда приложение использует API-интерфейсы ввода-вывода Windows для открытия удаленных файлов через UNC-пути или буквы дисков, отображенные на UNC-пути.

Маршрутизатор многосетевого доступа (MPR)

Windows WNet-функции позволяют приложениям (включая Центр управления сетями и общим доступом) подключаться к сетевым ресурсам, например к файловым серверам и принтерам, и просматривать различные общие пункты. Поскольку WNet API-интерфейс может быть вызван для работы с различными сетями с использованием различных транспортных протоколов, должно присутствовать программное обеспечение для отправки запроса к правильной сети и для разбора результатов, возвращенных удаленным сервером. На рис. 7.19 показано программное обеспечение перенаправления, отвечающее за выполнение этих задач.

Поставщик представляет собой программное обеспечение, которое учреждает Windows в качестве клиента удаленного сетевого сервера. Некоторые из операций, выполняемых поставщиком WNet, включают установку и прерывание сетевых соединений, а также поддержку сетевого вывода на печать. Встроенный

WNet-поставщик SMB включает в себя DLL-библиотеку, службу рабочей станции и редиректор. Другие сетевые производители нуждаются в поддержке только DLL-библиотеки и редиректора.

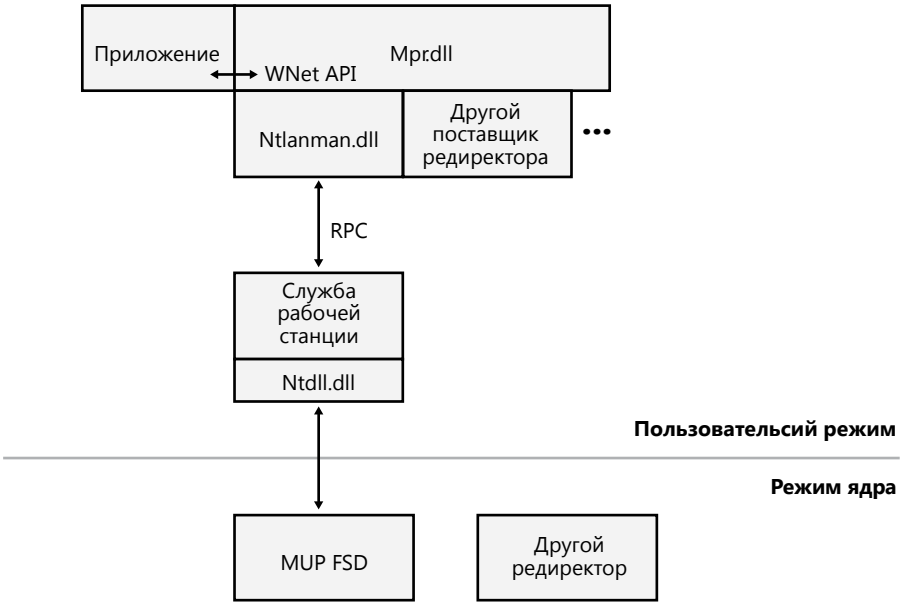


Рис. 7.19. Компоненты MPR

Когда приложение вызывает процедуру WNet, вызов передается непосредственно к MPR DLL. MPR принимает вызов и определяет, какой поставщик сети распознает тот ресурс, к которому запрашивается доступ. Каждый поставщик DLL, находящийся ниже MPR, предоставляет набор стандартных функций с общим названием интерфейс сетевого поставщика. Этот интерфейс позволяет MPR определить, к какой из сетей приложение пытается получить доступ, и направляет запрос к соответствующему программному обеспечению WNet-поставщика. Поставщиком службы рабочей станции SMB — SMB Workstation, является библиотека %SystemRoot%\System32\Ntlanman.dll, как определено параметром ProviderPath в разделе реестра HKLM\SYSTEM\CurrentControlSet\Services\LanmanWorkstation\NetworkProvider.

После вызова API-функцией WNetAddConnection2 или WNetAddConnection3 для подключения к удаленному сетевому ресурсу MPR проверяет параметр реестра HKLM\SYSTEM\CurrentControlSet\Control\NetworkProvider\HwOrder\ProviderOrder, чтобы определить, какие поставщики сети загружены. Он проводит их поочередный опрос в том порядке, в котором они перечислены в реестре, пока поставщик не распознает ресурс или пока не будут опрошены все доступные поставщики. Порядок опроса поставщиков ProviderOrder можно изменить, воспользовавшись диалоговым окном Дополнительные параметры (Advanced Settings), показанным на рис. 7.20. Получить доступ к диалоговому окну можно, набрав в поле поиска меню Пуск (Start) строку просмотр сетевых подключений и нажав клавишу Ввод. В результате появится диалоговое окно Сетевые подключения

(Network Connections). Нажмите клавишу Alt для отображения в диалоговом окне строки меню. Щелкните на раскрывающемся пункте меню Дополнительно (Advanced) и выберите пункт Дополнительные параметры (Advanced Settings), после чего щелкните на вкладке Порядок служб доступа (Provider Order).

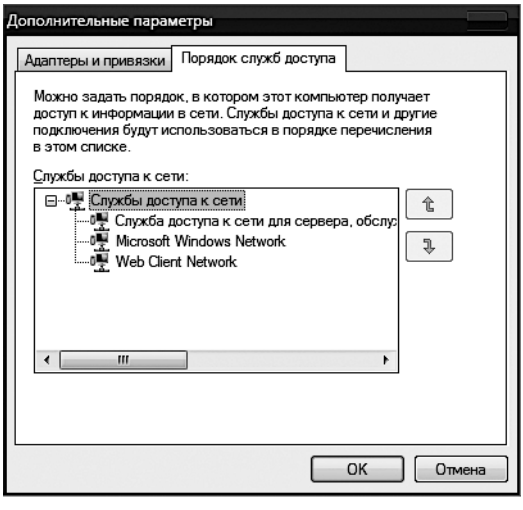


Рис. 7.20. Редактирование порядка следования поставщиков

Функция WNetAddConnection может также назначить удаленному ресурсу букву диска или имя устройства. При вызове ее с этими целями WNetAddConnection направляет вызов соответствующему поставщику сети. Поставщик, в свою очередь, создает объект символической ссылки в пространстве имен диспетчера объектов, отображающий букву диска, которая была определена, на редиректор (то есть на удаленный FSD) для этой сети.

На рис. 7.21 показан каталог Session 0 DosDevices, соответствующий LUID пользователя, осуществляющего отображение буквы диска, которое является местом хранения подключений к удаленным общим файловым ресурсам. Символическая ссылка, созданная поставщиками сети, чтобы служить в качестве связи между сетевым путем и соответствующим редиректором, полагается на MUP-поставщика. На рисунке показано, что этот MUP создает объект устройства \Device\LanmanRedirector, который представляет собой символическую ссылку на \Device\MUP (не показана на рисунке, поскольку находится в каталоге \Device) с дополнительным текстом, включенным в значение символической ссылки, который показывает MUP-редиректору, какой мини-редиректор соответствует букве диска. Каталог «\Global??» показывает вам буквы дисков, доступные сеансу системы, другие будут отображены в предназначенном для сеанса каталоге DosDevices.

Затем когда WNet или другой API вызывает диспетчер объектов для того, чтобы открыть ресурс на другой сети, диспетчер объектов использует объект устройства как отправную точку в удаленную файловую систему. Он вызывает метод анализа диспетчера ввода-вывода, связанный с объектом устройства для определения местонахождения редиректора FSD, который может обработать запрос.

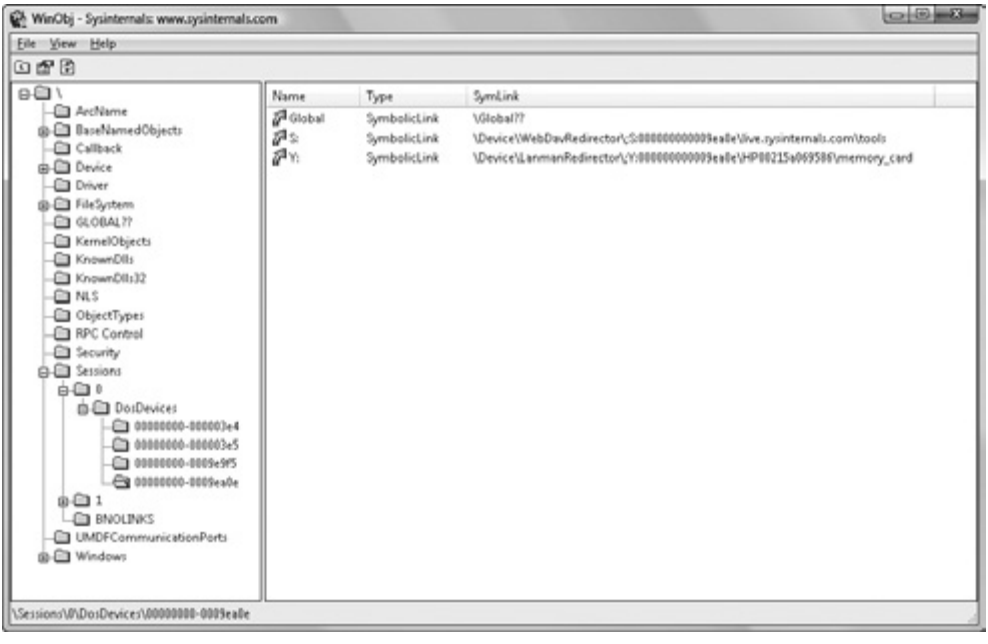


Рис. 7.21. Разрешение имени сетевого ресурса

Многосетевой UNC-поставщик (MUP)

Многосетевой UNC-поставщик (MUP, %SystemRoot%\System32\Drivers\mup.sys) является драйвером файловой системы, показывающим Windows удаленную файловую систему. Это единственное место, где драйверы фильтров файловой системы могут быть многоуровневыми для фильтрации любых запросов ввода-вывода к удаленным файловым системам (до выхода Windows Vista было много противоречивостей и сложностей относительно фильтрации удаленных файловых систем). MUP получает запросы ввода-вывода для доступа к удаленным файловым системам (через UNC-пути или буквы дисков, отображаемые на эти пути) и определяет, какой редиректор будет обрабатывать запрос. Термин *редиректор* используется потому, что он перенаправляет (redirects) запросы ввода-вывода на удаленную систему. Перед и дополнительно после вызова редиректора MUP будет вызывать любые зарегистрированные *заменители поставщиков*, которые могут предоставить кэширование файлов и перезапись пути.

MUP реализует то, что называется *префиксным кэшем*, представляющим собой список путей удаленной файловой системы (>]), обрабатываемых каждым редиректором. Возможно, что конкретный префикс могут обрабатывать сразу несколько редиректоров, поэтому в реестре (HKLM\System\CurrentControlSet\Control\NetworkProvider\Order\ProviderOrder) имеется список, содержащий разделенный запятыми список приоритетного порядка, в котором MUP направляет запросы редиректорам. Этот список также используется для загрузки поставщиков. В разделе ProviderOrder имеются два подраздела (HwOrder и Order),

которые в параметре ProviderOrder содержат одинаковую информацию. Обычно параметр имеет следующий вид:

```
ProviderOrder     REG_SZ     RDPNP,LanmanWorkstation,webClient
```

Каждая запись определяет имя службы в HKLM\System\CurrentControlSet\Services, где можно найти еще один подраздел NetworkProvider. Например, в разделе HKLM\System\CurrentControlSet\Services\RDPNP\NetworkProvider находятся следующие параметры:

```
DeviceName      REG_SZ          \Device\RdpDr
DisplayName     REG_EXPAND_SZ   @%systemroot%\system32\drprov.dll,-100
Name            REG_SZ          Microsoft Terminal Services
ProviderPath    REG_EXPAND_SZ   %SystemRoot%\System32\drprov.dll
```

Параметр DeviceName содержит имя, присвоенное объекту устройства редиректора режима ядра. Параметр DisplayName содержит официальное имя поставщика. (Оно может быть либо строкой, либо, как показано в примере, местом нахождения строки в разделе ресурсов DLL-библиотеки.) Параметр Name содержит имя, которое будет выведено командой net use для определения того, какой редиректор владеет конкретным накопителем. Параметр ProviderPath указывает путь к месту хранения DLL поставщика.

ПРИМЕЧАНИЕ

Не все редиректоры выводятся или должны выводиться в порядке следования поставщиков. (Обычно в списке будут показаны только RDPNP, LanmanWorkstation, webclient.) Приоритет редиректоров, не перечисленных в реестре, следует за теми, которые перечислены в порядке убывания, а затем основывается на том порядке, в котором мини-редиректор зарегистрирован с помощью MUP через функцию FsRtlRegisterUncProviderEx и функцию RxRegisterMinirdr.

Компоненты префикса (имя сервера и имя общего ресурса), которые требуются редиректору, изменяются, большинство редиректоров обычно требуют в UNC-пути как имя сервера, так и имя общего ресурса (\\<*имя_сервера*>\<*имя_общего_ресурса*>[\<*путь*>]). Например, для пути \\Server\Users\Brian\Documents редиректор может потребовать префикс \\Server\Users, что заставит MUP направить все запросы, содержащие этот префикс, этому конкретному редиректору, например \\Server\Users\David\Documents\Chapter7.doc, но путь с префиксом \\Server\Backups должен быть разрешен через запрос редиректоров в порядке их приоритета. Если редиректор требует префикс, содержащий только имя сервера (например, \\Server), MUP отправляет редиректору запросы ко всем общим ресурсам (например, \\Server\Users, \\Server\WebDAV и т. д.) этого сервера.

MUP использует имена, найденные в ProviderOrder, для поиска имени устройства, реализующего редиректор, путем поиска в HKLM\System\CurrentControlSet\Services\<*имя_редиректора*>\NetworkProvider\DeviceName. Значение параметра DeviceName содержит символическую ссылку, указывающую на MUP, например \Device\MUP\;LanmanRedirector. (Точка с запятой показывает, что это «открытая целенаправленность», означающая, что MUP не будет заглядывать в префиксный кэш.)

Взаимоотношения между MUP и другими компонентами, являющимися частью удаленной файловой системы, показаны на рис. 7.22.

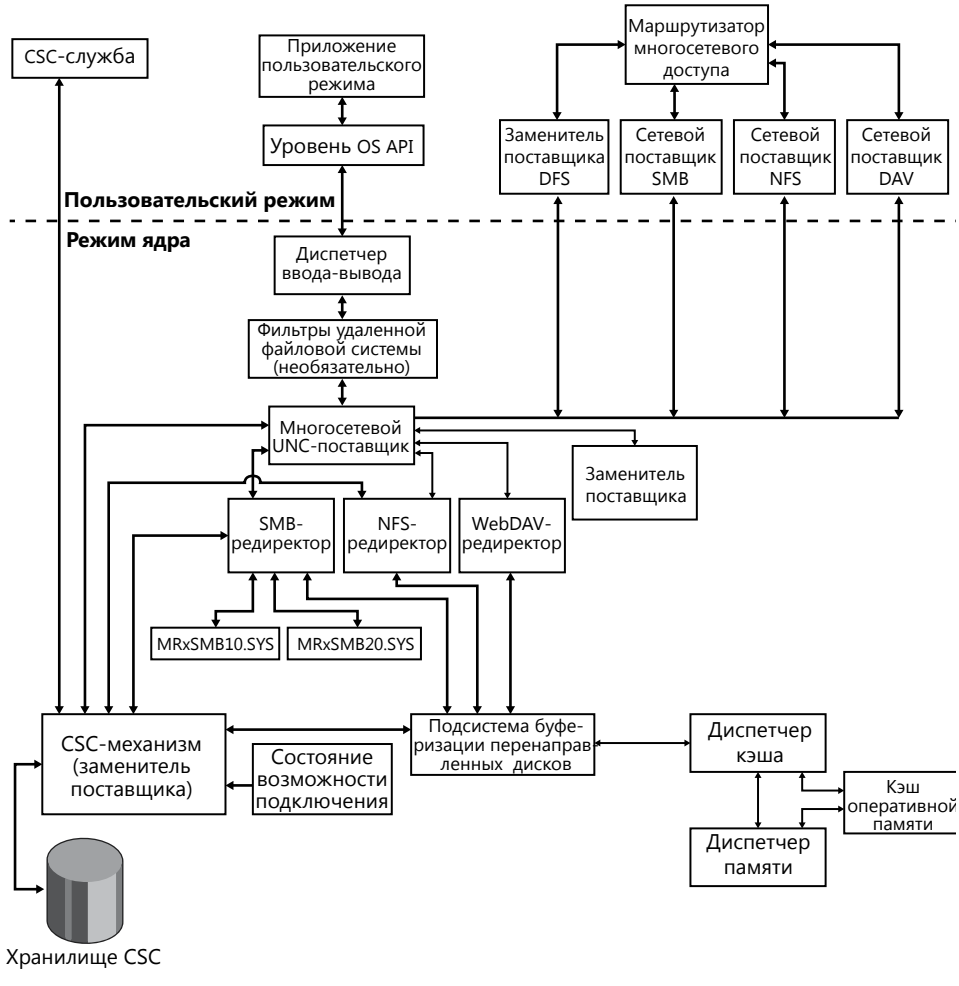


Рис. 7.22. Архитектура MPR и UNC

Заменители поставщиков

До выхода Windows Vista кэширование удаленных файловых систем (механизм Offline files) было реализовано внутри мини-редиректора SMB, а клиент DFS-N (Distributed File System Namespace — пространство имен распределенной файловой системы) был реализован внутри MUP. Нужен был единый кэш, поэтому архитектура удаленной файловой системы для Windows Vista была переработана. Клиент DFS-N был перемещен в отдельный драйверный компонент, известный как заменитель поставщика MUP, и поддержка Offline files стала осуществляться отдельным драйвером, работающим и как мини-редиректор, и как заменитель поставщика. В настоящее время имеется два заменителя поставщиков:

- Автономные файлы — Offline Files (%SystemRoot%\System32\Drivers\csc.sys), — которые определяют, должен ли запрошенный файл быть кэширован локально, или был ли он уже кэширован. Для службы Автономные файлы жестко задан самый высокий приоритет заменителя.
- Клиент распределенной файловой системы — Distributed File System Client (%SystemRoot%\System32\Drivers\dfsc.sys), — который определяет, должен ли быть изменен (переписан) путь к запрошенному файлу, чтобы указывать на другой сервер или общий ресурс. (Суть DFS-N заключается в том, что он собирает один или несколько сетевых общих ресурсов в едином пространстве имен.) Для DFSC жестко задан второй по значимости приоритет заменителя.

Может показаться, что наличие заменителей в пути между MUP и редиректорами приведет к снижению производительности, но Offline files не обрабатывает пути, которые не разрешены для автономного доступа, и после отказа от пути MUP не будет для Offline files заниматься дальнейшим вводом-выводом, направляемым с использованием пути. Точно так же DFS не обрабатывает не относящиеся к нему пути.

Список заменителей жестко задан, поэтому MUP не поддерживает добавление дополнительных заменителей.

Редиректор

Сетевой редиректор состоит из установленных в системе программных компонентов, которые поддерживают доступ к различного типа ресурсам на удаленных системах, используя различные сетевые файловые протоколы. Типы поддерживаемых редиректором ресурсов зависят от редиректора и возможностей протокольной системы. Фактически все редиректоры поддерживают UNC-имена, которые разрешают удаленное общее использование таких ресурсов, как файлы, принтеры, именованные каналы и почтовые слоты (хотя редиректор может отказаться от поддержки каналов и почтовых слотов, продолжая поддерживать принтеры и файлы). Редиректоры, поставляемые в Windows, включают следующие компоненты:

- DLL-библиотеку, загружаемую MPR в пользовательском режиме для выполнения операций, не связанных с файлами, например для определения возможностей поставщика сети, перебора удаленных сетевых ресурсов, входа в удаленную сеть и монтирования удаленных сетевых ресурсов;
- драйвер режима ядра, известного как миниредиректор, который импортирует экспортируемый драйвер RDBSS (Redirected Drive Buffering SubSystem — подсистемы буферизации перенаправленных дисков) (%SystemRoot%\System32\Drivers\rdbss.sys). Мини-редиректор обслуживает файловые запросы ввода-вывода, направляемые на удаленные системы.

Некоторым редиректорам требуется один или несколько дополнительных компонентов:

- Служебный процесс для помощи DLL-библиотеке и возможного хранения конфиденциальной информации или информации, которая является глобальной для всех клиентских приложений, использующих конкретную сеть или

общий ресурс. Например, служба рабочих станций (выполняющаяся в процессе SVCHOST) отслеживает отображение буквы диска на путь *сервер**общий_ресурс*.

- Драйвер сетевого протокола, который реализует устаревший интерфейс транспортного драйвера — Transport Driver Interface (TDI) — на своей верхушке, требуется в том случае, если редиректор использует сетевой протокол, не поставляемый Windows. (По сути это означает любой другой, кроме TCP/IP.) Такой драйвер протокола отвечает за реализацию обмена данными с удаленной системой.
- Служебный процесс, помогающий редиректору. Например, редиректор WebDav перенаправляет операции доступа к файлам службе пользовательского режима WebClient, которая, в свою очередь, выдает фактические запросы к сетевому протоколу WebDav, используя HTTP API-интерфейс.

Редиректор представляет ресурсы, которые прикреплены к удаленным системам таким образом, будто они были прикреплены к локальной системе. В Windows нет специальных API-интерфейсов файлового ввода-вывода, требуемых для доступа к ресурсам удаленной системы. При обращении к ресурсу приложение обычно не знает и даже не заботится о том, где находится этот ресурс, на локальной или же на удаленной системе. Название «редиректор» используется потому, что он перенаправляет (redirects) операции файловой системы на удаленную систему и возвращает приложению ответ от удаленной системы.

Все редиректоры, поставляемые с Windows, реализованы с использованием мини-редиректорной архитектуры, где код, имеющий отношение к протоколу реализован в драйвере мини-редиректора, который импортирует RDBSS-библиотеку. RDBSS реализована в виде драйвера класса, и мини-редиректоры похожи на драйверы портов. RDBSS регистрируется с помощью MUP путем вызова функции FsRtlRegisterUncProviderEx.

Когда мини-редиректор регистрируется с RDBSS через функцию RxRegisterMiniRdr, RDBSS в свою очередь регистрируется с MUP путем вызова функции FsRtlRegisterUncProviderEx. MUP направляет запросы (IRP-пакеты) подсистеме RDBSS, которая выполняет общую для всех удаленных файловых систем обработку, а затем выдает упрощенные запросы через функции обратного вызова, которые мини-редиректоры связали с тем, с чем они были зарегистрированы. RDBSS предоставляет общие функциональные средства, такие как структуру данных и модель блокировки, объединение в систему диспетчера кэша и диспетчера памяти и обработку IRP-пакетов. Это упрощает реализацию мини-редиректоров и существенно сокращает объем создаваемого и отлаживаемого кода.

Поскольку RDBSS интегрирована с диспетчером кэша, мини-редиректоры RDBSS могут не видеть напрямую запросы на чтение и запись в отношении буферированных дескрипторов (дескрипторов, открытых без указания флага FILE_FLAG_NO_BUFFERING при вызове API-функции CreateFile). Изменения кэшируются диспетчером кэша на локальной системе, пока не возникнет потребность в их записи на удаленную систему. Это улучшает показатель времени отклика и экономит сетевой трафик за счет накопления данных для записи и исключения случаев повторного чтения. RDBSS полагается на мини-редиректор, сообщая ему, когда можно будет безопасно кэшировать данные для чтения

и (или) записи. Например, для управления кэшированием мини-редиректор SMB использует уступающие блокировки — opportunistic locks (более известные как oplocks). Блокировка oplock является механизмом согласования кэша, позволяющим потребителям файловой системы динамически изменять свои состояния кэширования для заданного файла или потока данных, поддерживая согласованность кэша между несколькими конкурирующими пользователями файла. Если файл (или поток) в данный момент не открыт для чтения или записи другим средством доступа (либо локально, либо удаленно), клиент может локально кэшировать чтения, записи и блокировку диапазона байтов. Если файл открыт другими пользователями, но не находится в процессе записи, записи и блокировки локально кэшироваться не будут, но чтения по-прежнему будут кэшироваться.

Мини-редиректоры

Миниредиректор реализует протокол, необходимый для контакта с удаленной системой и получения доступа к ее общим ресурсам. Мини-редиректор пытается получить доступ к удаленным ресурсам как можно более прозрачно для локального клиентского приложения. Например, если возникают проблемы с работой сети, редиректор может несколько раз повторить попытку запроса, прежде чем вернуть клиентскому приложению ошибку.

В систему Windows включены несколько мини-редиректоров:

- RDPDR (Remote Desktop Protocol Device Redirection — драйвер перенаправления устройств протокола удаленного рабочего стола), который позволяет получить доступ из системы сервера терминалов к клиентским системным файлам и принтерам (%SystemRoot%\System32\Drivers\rdpdr.sys).
- Протокол SMB (Server Message Block — блок сообщений сервера), который является стандартной удаленной файловой системой, используемой Windows (%SystemRoot%\System32\Drivers\MRxSMB.SYS). Его иногда называют CIFS, или Common Internet File System — общий протокол доступа к интернет-файлам. MRxSMB.SYS загрузит подчиненные редиректоры (SubRedirectors), рассматриваемые в следующем разделе.
- Протокол WebDAV (Web Differencing and Versioning), позволяющий получить доступ к файлам по HTTP(S)-протоколу (%SystemRoot%\System32\Drivers\MRxDAV.SYS).
- Почтовый слот MailSlot (часть MRxSMB.SYS). Почтовые слоты обрабатываются совсем не так, как именованные каналы. Заменители для отправки ввода-вывода почтовому слоту не вызываются, и кэширование префиксов не используется. (Все пути, в которых в качестве общей части имеется подстрока «mailslot», нацеливаются непосредственно к мини-редиректору почтовых слотов.) Должно быть не более одного мини-редиректора почтового слота, и в настоящее время он зарезервирован для SMB-редиректора.
- Сетевая файловая система — Network File System (NFS) — является дополнительным компонентом, который раньше был установлен со службами для Unix — Services For Unix (SFU), а теперь стал дополнительным компонентом Windows, который может быть установлен с помощью панели управления

Программы и компоненты (Programs and Features)[1]. Поддерживается протокол версий 2 и 3.

Механизм Автономные файлы — Offline Files, рассматриваемый в одном из следующих разделов, дополнительно позволяет проводить кэширование диска и получать автономный доступ к файлам, доступным через SMB-протокол. Механизм Автономные файлы также регистрируется в качестве заменителя поставщика MUP.

Протокол блока сообщений сервера и подчиненные редиректоры

Протокол блока сообщений сервера — Server Message Block (SMB) — является основным протоколом удаленного доступа к файлам, используемым клиентами и серверами Windows, и относится к 1980-м годам. SMB версии 1.0 (на которую ссылаются, как на SMB) была разработана для работы с дружественной LAN-средой, где скорость передачи данных была обычно 10 Мбайт/с и где никто не покушался на ваши данные. Для выполнения многих задач общего назначения необходимы серии синхронных сообщений между клиентом и сервером. Особый расчет на WAN-сети не делался, поскольку эти сети в то время еще не были достаточно развиты. В 1996 году SMB был представлен на рассмотрение группы IETF как общий протокол доступа к интернет-файлам — Common Internet File System (CIFS). Microsoft создала описание протокола CIFS/SMB в документах по протоколам MS-CIFS и MS-SMB.

Протокол SMB 2.0 был выпущен в Windows Vista и Windows Server 2008 и представлял собой полную переделку основного протокола доступа к уделенным файлам для Windows. SMB 2.0 предоставляет ряд улучшений SMB, в числе которых:

- существенное упрощение: количество наборов операций снизилось с более чем ста до всего девятнадцати;
- снижение «болтливости» протокола, что делает его более подходящим для запуска по WAN-сетям, у которых обычно более продолжительное время ожидания и менее высокая пропускная способность, чем у LAN-сетей;
- составные запросы позволяют отправлять сразу несколько запросов в одном сетевом пакете;
- конвейеризация запросов, позволяющая нескольким запросам и данным отправляться до получения ответа на предыдущий запрос (это известно также как поток управления, основанный на кредитах);
- более высокие объемы чтений и записей;
- кэширование свойств папок и файлов;
- улучшение алгоритма подписи сообщений (HMAC SHA-256 заменил MD5);
- улучшение масштабируемости совместного использования файлов;
- улучшенная работа с преобразованием сетевых адресов — Network Address Translation (NAT);
- поддержка символических ссылок.

[1] Щелкните на пункте Включение или отключение компонентов Windows (Turn Windows Features On Or Off), а затем установите флажок Службы для NFS (Services For NFS).

Версия 2.1 протокола SMB (выпущенного вместе с Windows 7 и Windows Server 2008/R2) является минорной версией, документированной в спецификации протокола MS-SMB2. Она добавляет следующие усовершенствования:

- Новая лизинговая модель уступающей блокировки — opportunistic lock (oplock), которая дает больше возможностей по кэшированию файлов и дескрипторов, не требуя при этом внесения изменений в существующие приложения.
- Поддержка еще более крупных блоков передачи (больших MTU), от предыдущего максимума в 64 Кбайт до 1 Мбайт (по умолчанию, но в реестре можно настроить вплоть до 8 Мбайт).

Для поддержки обратной совместимости с SMB-серверами клиент SMB2 использует существующие механизмы настройки подключений SMB, а затем извещает о поддержке более новой версии протокола. Мини-редиректор SMB содержит все функциональные возможности, которые являются общими для всех версий протокола, с отдельным подчиненным редиректором, реализующим каждый из вариантов SMB-протокола. Клиент SMB2 устанавливает соединение и отправляет запрос на согласование SMB, в котором содержатся оба поддерживаемых диалекта, как SMB, так и SMB2. Если сервер поддерживает SMB2, он откликается с согласованным ответом SMB2, и клиент вручает соединение подчиненному редиректору SMB2. С этого момента все сообщения в соединении передаются по протоколу SMB2. Если сервер не поддерживает SMB2, он откликается с переговорным ответом SMB, и клиент вручает соединение подчиненному редиректору SMB1:

- В файле %SystemRoot%\System32\Drivers\MRxSMB.sys реализованы общие части.
- В файле %SystemRoot%\System32\Drivers\MRxSMB10.sys реализован протокол SMB 1.
- В файле %SystemRoot%\System32\Drivers\MRxSMB20.sys реализован протокол SMB 2.

Пространство имен распределенной файловой системы

Пространство имен распределенной файловой системы — Distributed File System Namespace (DFS-N) — является объединением пространств имен и свойством доступности Windows. По мере роста организации возникает тенденция к росту количества файловых серверов, и пользователям становится все труднее искать нужные файлы, потому что они могут быть разбросаны по большому количеству различных серверов с полностью не связанными друг с другом именами. DFS-N позволяет администратору создать новый общий файловый ресурс (также известный как корень или пространство имен), объединяющий несколько общих файловых ресурсов одного и того же или разных серверов в единое пространство имен. Предположим, например, что в Aura Corporation имелись следующие общие ресурсы: \\Development\Projects, \\Accounting\FY2012 и \\Marketing\CoolStuff. Эти общие ресурсы могут быть представлены пользователям через

пространство имен DFS-N \\Aura\Teams, содержащее DFS-N-ссылки с именами \\Aura\Teams\\Aura\Development, \\Aura\Teams\Accounting и \\Aura\Teams\Marketing. Перенаправление клиента, обращающегося к пути \\Aura\Teams\Marketing, на реальный путь к общему ресурсу \\Marketing\CoolStuff осуществляется незаметно для пользователя. В данном примере \\Marketing\CoolStuff является целью ссылки \\Aura\Teams\Marketing. Фактически, цели ссылок могут ссылаться на пути ниже корня общего ресурса, как в примере \\Marketing\CoolStuff\Presentations.

Другие преимущества, предоставляемые DFS-N, заключаются в избыточности и в перенаправлении в известное место. Еще одной важной возможностью DFS является доступность, обеспечиваемая через свойство, известное как тиражирование, или репликация DFS — DFS Replication (DFSR). Тиражирование дает два преимущества: высокую доступность в случае отказа и сбалансированность нагрузки. При географическом расширении организации доступ к файловым серверам из удаленных офисов с использованием соединений по глобальной сети — wide area network (WAN) — может быть медленным и неэффективным. Администратор может создать тиражированную версию файлового сервера в удаленном офисе, предоставляя высокоскоростной доступ к файлам тем пользователям, которые находятся в удаленном офисе. DFS-N-ссылка, например \\Aura\Teams\Accounting из предыдущего примера, может иметь несколько целей, связанных с этой ссылкой, например \\AccountingEurope\FY2012 и \\AccountingUS\FY2012. В таком случае DFS-N-сервер возвращает клиенту упорядоченный список доступных целевых серверов (используя информацию сетевого узла Active Directory), где элементы в списке выстроены в порядке, позволяющем клиенту получить сначала доступ к ближайшей цели. Если доступ к одной цели ссылки оказывается безуспешным, DFS-N пробует воспользоваться следующей доступной целью, если это возможно. Когда DFS-N-ссылка имеет несколько целевых общих ресурсов, цели, как правило, содержат одни и те же данные, поскольку клиент, обращающийся к пространству имен, получит в данный момент доступ только к одной из целей. Это может быть выполнено с использованием тиражирования DFS Replication (DFS-R), рассматриваемого в следующем разделе. Реализация DFS-N на стороне сервера состоит из Windows-службы (%SystemRoot%\System32\Dfssvc.exe) и драйвера устройства (%SystemRoot%\System32\Drivers\Dfs.sys). DFSSVC-служба отвечает за экспорт DFS-интерфейсов управления топологией и обслуживает DFS-топологию как в реестре (на системах, не использующих Active Directory systems), так и в Active Directory. DFS-драйвер выполняет поиски в топологии при получении пользовательского запроса, касающегося ссылки, чтобы иметь возможность направить клиента к общему ресурсу, в котором размещается запрошенный файл.

На стороне клиента поддержка DFS-N реализована в заменителе драйвера поставщика MUP (%SystemRoot%\System32\Drivers\Dfsc.sys) и в поставщике MPR/WNet, который реализован в библиотеке %SystemRoot%\System32\Ntlanman.dll. Драйвер клиента распределенной файловой системы — Distributed File System Client (DFSC) — отвечает за определение, является ли UNC-путь пространством имен DFS, и если является, он преобразует указанный путь в имя одного или нескольких целевых общих ресурсов. Связь с DFS-N-серверами организуется с помощью SMB-редиректора. DFS-N-клиент является лишь частью

пути ввода-вывода при создании или открытии файла или каталога. Как только имя целевого общего ресурса возвращается MUP, DFSC не участвует в последующем файловом вводе-выводе.

Протоколы DFS-N описаны в документах протоколов MS-DFSC и MS-DFSNM.

Репликация распределенной файловой системы

Репликация распределенной файловой системы — Distributed File System Replication (DFS-R) — предоставляет эффективно использующую полосу пропускания, асинхронную репликацию изменений файловой системы с несколькими главными копиями на нескольких серверах. Кроме решения задач общего назначения, то есть репликации файловой системы (например, организации синхронного хранения данных на нескольких общих ресурсах, на которые указывают цели DFS-N-ссылки), DFS-R также используется для репликации каталога доменных контроллеров \SYSVOL, который служит в Windows местом хранения доменными контроллерами сценариев входа в систему и файлов групповой политики[1]. Поскольку DFS-R поддерживает репликацию с несколькими главными копиями (multimaster replication), изменения файловой системы может произойти на любом сервере практически одновременно, и DFS-R автоматически уладит конфликты и обеспечит синхронизацию содержимого файловой системы.

Основной единицей DFS-репликации служит папка DFS-репликации, которая является деревом каталогов, чье содержимое будет синхронизировано между несколькими серверами в соответствии с административно определенным расписанием и топологией репликации. Расписания репликации позволяют администраторам ограничить репликационную активность определенными окнами времени или ограничить полосу пропускания, используемую DFS-R.

Репликационные топологии позволяют администраторам определять сетевые соединения между набором серверов (называемым репликационной группой). Поддерживаются произвольные топологии, включая такие общепринятые топологии, как кольцо, звезда или сетка. Конфигурация репликационной топологии хранится в Active Directory.

Репликации могут подвергаться только каталоги на NTFS-томах, поскольку DFS-R для обнаружения изменений содержимого реплицируемой папки полагается на NTFS USN-журнал.

Для сохранения пропускной способности сети в DFS-R используются несколько технологий, подгоняя репликацию под работу с WAN-сетями, у которых может быть высокий уровень задержки и низкая пропускная способность. Удаленное разностное сжатие — Remote Differential Compression (RDC) — позволяет DFS-R обнаруживать и реплицировать только те части файла, которые подверглись изменению, а не весь файл. DFS-R также сжимает содержимое перед его отправкой удаленному партнеру, предоставляя дополнительную экономию по-

[1] Групповая политика позволяет администраторам определять политики использования и безопасности для компьютеров, принадлежащих домену.

лосы пропускания. В ассортиментных позициях Enterprise или Datacenter DFS-R использует расширенную версию RDC, которая называется RDC Similarity, для предоставления еще большей экономии полосы пропускания, если содержимое изменяется в реплицируемой папке на сервере А и фрагменты измененного содержимого аналогичны фрагментам любого файла в реплицируемой папке партнерского сервера Б, сервер Б довольствуется тем, что обновляет содержимое за счет локальных аналогичных фрагментов из подобных файлов, а не загружает все измененное содержимое с сервера А.

Новые возможности для DFS-R в Windows Server 2008 R2 включают поддержку кластеризованных копий и копий, предназначенных только для чтения.

Репликация DFS-R реализована в виде Windows-службы (%SystemRoot%\System32\DfsrS.exe), использующей для связи между своими собственными экземплярами, запущенными на разных компьютерах, аутентифицированные RPC с шифрованием. Существует также WMI-интерфейс для настройки и управления службой, мини-фильтр файловой системы, используемый для защиты от изменения копий, предназначенных только для чтения, и DLL-библиотека кластерного ресурса для интеграции с MSCS. Протокол DFS-R описан в спецификации MS-FRS2.

Автономные файлы

Служба Автономные файлы (Offline Files) — внутри системы также называется кэширование на стороне клиента (client-side caching, или CSC) — открыто кэширует файлы из удаленной системы (файлового сервера) на локальной машине, чтобы эти файлы были доступны, когда локальная машина не подключена к сети. Служба Автономные файлы кэширует файлы, относящиеся к удаленным файлам, доступным через SMB-протокол. Файлы могут кэшироваться пользователями простым щелчком правой кнопкой мыши на удаленном файле, папке или диске с последующим выбором пункта Always Available Offline (Всегда быть доступным автономно), прикрепляя тем самым выбранные файлы к кэшу. Кэшируемые элементы могут быть просмотрены в Центре синхронизации (Sync Center) Панели управления. Кэширование может быть также указано административным образом с помощью групповой политики.

В системе имеется единый кэш службы Автономные файлы, который совместно используется всеми пользователями системы. Все кэшированные файлы хранятся в каталоге, защищенном ACL-списком, в качестве которого по умолчанию служит каталог %SystemRoot%\CSC. По своему решению вы можете зашифровать файлы в кэш службы Автономные файлы (эта возможность доступна при переходе в Панель управления (Control Panel), затем в Центр синхронизации (Sync Center), после чего следует щелкнуть на пункте Управление автономными файлами (Manage Offline Files), щелкнуть на вкладке Шифрование (Encryption) и щелкнуть на кнопке Зашифровать (Encrypt)). Доступ к кэшу разрешен только путем использования инструментария службы Автономные файлы и COM API-интерфейса IOfflineFilesXxx. Самый простой способ проверить содержимое кэша заключается в использовании интерфейса Панели управления Центр синхронизации (Sync Center) (Управление автономными файлами (Manage Offline Files) ▸ Просмотреть автономные файлы (View Your Offline Files)).

Служба Автономные файлы понимает два типа объектов:

- **Files** (файлы). Включает файлы, папки и символические ссылки. Кэширование на NTFS-уровне не производится, поэтому не все файловые NTFS-атрибуты кэшируются или являются кэшируемыми. Кэшируемые атрибуты включают стандартные файловые атрибуты Win32 (метаданные), такие как имя, ACL-список и содержимое — кэшироваться будет только файловый (безымянный) поток данных.
- **Scope** (область видимости). Область видимости является частью пространства имен, которая соответствует физическому общему ресурсу. В пространстве имен DFS корнем области видимости служит объект, на который указывает DFS-ссылка. Он может содержать дополнительные DFS-ссылки на другие области видимости. Если DFS не используется, то область видимости и общий ресурс — это одно и то же.

Служба Автономные файлы не поддерживает полную NTFS-семантику для кэшированных файлов и имеет следующие ограничения:

- Служба Автономные файлы не кэширует изменяющиеся потоки данных, которые по этой причине не доступны в автономном режиме. В режиме подключения к сети доступ к изменяющимся потокам данных работает по той причине, что запросы на ввод-вывод для потоков поступают непосредственно на сервер.
- Служба Автономные файлы не кэширует расширенные атрибуты — Extended Attributes (EA). Вследствие этого, если файл, содержащий EA-атрибуты, кэшируется и кэшированная версия изменяется в то время, когда сервер отключен, любые EA-атрибуты на сервере удаляются, когда изменения записываются на сервер.

В службу Автономные файлы (рис. 7.23) входят следующие компоненты:

- Агент пользовательского режима (%SystemRoot%\System32\cscsvc.dll), работающий в качестве службы в процессе SVCHOST. Эта служба отвечает в первую очередь за обеспечение синхронизации между кэшем и удаленными файловыми системами. Она также реализует COM-интерфейсы, используемые для взаимодействия с кэшем службы Автономные файлы.
- Драйвер удаленной файловой системы (%SystemRoot%\System32\Drivers\csc.sys), работающий как заменитель поставщика MUP и как мини-редиректор. Этот драйвер отвечает за управление тем, куда отправляются запросы ввода-вывода, к кэшу или к удаленной файловой системе. Драйвер также реализует локальный кэш, обновляя по мере необходимости кэшированные данные на основе увиденных запросов ввода-вывода.
- DLL-библиотека расширения обозревателя Explorer (%SystemRoot%\System32\cscui.dll) для выбора тех файлов, папок или дисков, которые прикрепляются к кэшу службы Автономные файлы, и для отображения накладок на значки для определения автономных (кэшированных) файлов. Библиотека CSCUI имеет ссылку на библиотеку %SystemRoot%\System32\cscobj.dll, которая предоставляет интерфейс службе Автономные файлы.

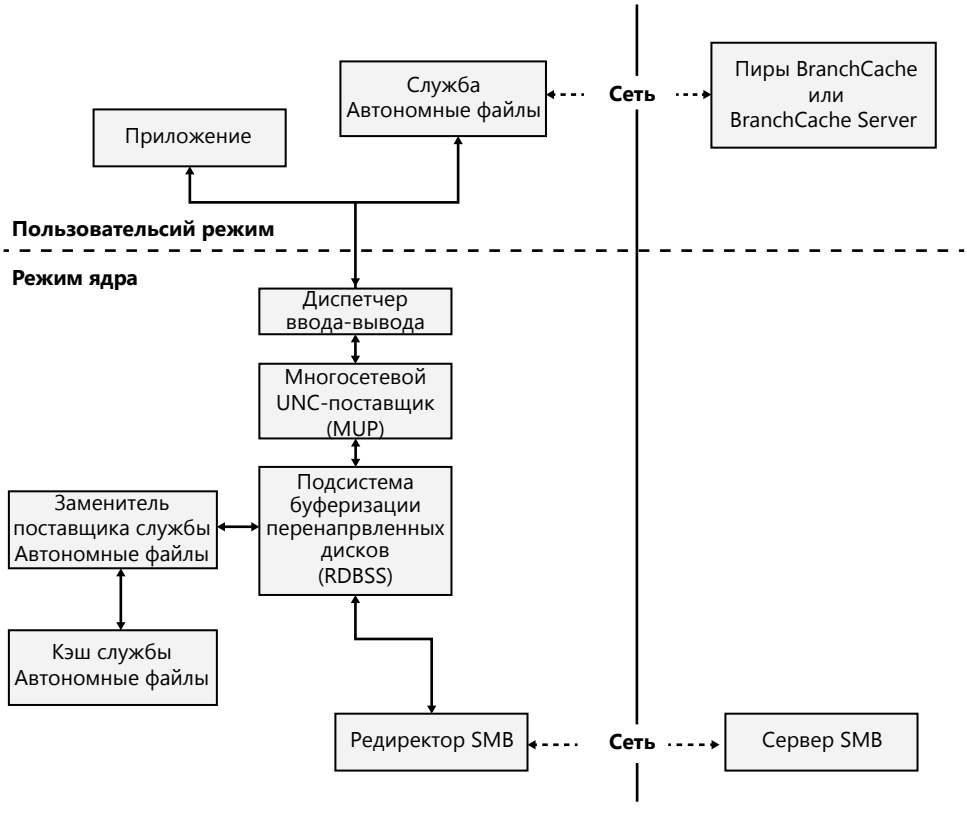


Рис. 7.23. Архитектура механизма Автономные файлы

- DLL-библиотека (%SystemRoot%\System32\cscapi.dll), в которой содержатся широкодоступные Win32 API-функции для взаимодействия со службой Автономные файлы из приложений.
- Встроенный в процесс COM-объект (%SystemRoot%\System32\cscobj.dll), используемый в приложении клиентами COM API-интерфейсов службы Автономные файлы.

Режимы кэширования

Служба Автономные файлы имеет пять режимов кэширования. Режим для объекта зависит от состояния подключения объекта, которое определяется тем, имеет или нет локальная система сетевое подключение к файловому серверу.

Online — режим подключения к сети

Для объектов, кэшируемых службой Автономные файлы, этот режим используется по умолчанию. В этом режиме сервер доступен. Операции метаданных файловой системы и операции записи следуют на сервер, и состояние кэша обновляется по мере надобности. Операции чтения обслуживаются из кэша.

При работе в режиме подключения к сети служба Автономные файлы пытается кэшировать данные, только если SMB-клиент получил от файлового сервера как минимум привилегии на чтение из кэша.

Offline (Slow Connection) — автономный режим (режим медленного подключения)

Чтобы изолировать пользователя от отклонений в работе сети, служба Автономные файлы переходит в автономный режим (режим медленного подключения) — Offline (Slow Connection), когда производительность сети доходит до настроенного значения задержки при медленном подключении или определенного снижения порога пропускной способности. В Windows 7 пороговое значение задержки при медленном подключении настроено на 80 миллисекунд (мс). Пороговые значения задержки и снижения пропускной способности могут управляться через редактор локальной групповой политики (%SystemRoot%\gpedit.msc) посредством пункта Настроить режим медленного подключения (Configure Slow-Link Mode).

При работе в данном режиме все операции файловой системы обслуживаются кэш-структурами службы Автономные файлы. Обратная синхронизация данных на сервере проводится по умолчанию каждые шесть часов, но эта частота синхронизации может управляться через редактор локальной групповой политики посредством пункта Настроить фоновую синхронизацию (Configure Background Sync).

Служба Автономные файлы периодически проверяет сетевую производительность общих ресурсов в кэше этой службы. Если задержки сети уменьшаются и их значение становится меньше половины порогового значения, заданного для медленного подключения, пользователь будет переведен назад на работу в режиме подключения к сети.

Поведение системы при медленном подключении может быть настроено через редактор локальной групповой политики (рис. 7.24).

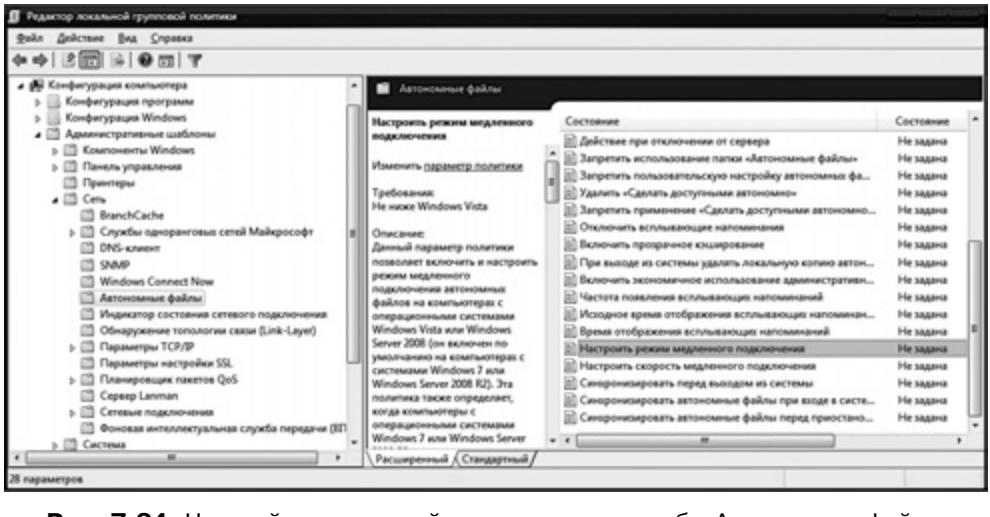


Рис. 7.24. Настройки групповой политики для службы Автономные файлы

Offline (Working Offline) — Режим работы вне сети

Пользователь может заставить клиентскую систему работать автономно, щелкнув в Explorer на кнопке Режим работы вне сети (Work Offline). При работе в этом режиме все операции файловой системы удовлетворяются из кэша. В этом режиме через политику Настроить фоновую синхронизацию (Configure Background Sync) может быть включена периодическая фоновая синхронизация данных, но по умолчанию она отключена. Если пользователь опять хочет работать в режиме подключения к сети, он должен щелкнуть в Explorer на кнопке Работать в сети (Work Online).

Offline (Not Connected) — Режим работы вне сети при отсутствующем подключении

Кэшированный объект находится в режиме работы вне сети при отсутствующем подключении, когда сервер недоступен. Переход на автономную работу вполне очевидно удовлетворяется благодаря кэшу службы Автономные файлы, без уведомления об этом приложения. Когда сетевое подключение к серверу восстанавливается, любые изменения, записанные в файл, синхронизируются с данными на сервере агентом службы Автономные файлы. Если, пока работа с файлом шла в автономном режиме, он изменился как на клиентской стороне, так и на удаленной системе, конфликт должен быть разрешен пользователем через Центр синхронизации (Sync Center).

Offline (Need to Sync) — Вне сети (необходима синхронизация)

Когда пользователь возвращается назад в режим работы в сети после внесения изменений в версию файла, находящуюся в локальном кэше, файл будет иметь статус «вне сети» (необходима синхронизация» — Offline (Need to Sync) до тех пор, пока изменения не будут синхронизированы с данными на сервере. Служба Автономные файлы продолжает держать работу пользователя вне сети для находящихся в этом статусе файлов, пока не завершится синхронизация, чтобы гарантировать, что перед пользователем находится непротиворечивое содержимое файлов, включая те изменения, которые были сделаны при работе вне сети.

Призраки

Когда файлы выбраны доступными для работы в автономном режиме, они должны быть скопированы с сервера на клиентскую машину. Пока передача не завершится, клиенту будут видимы не все файлы. Ситуация, при которой сервер внезапно оказывается недоступен, а пользователь пытается получить доступ к файлу до того, как он попадает в кэш, может вызвать у пользователя недоумение. Именно для такого случая служба Автономные файлы создает при включенном кэшировании *призраки* файлов и каталогов сервера в кэше. Призраки являются маркерами для файлов и каталогов, которые еще не были скопированы и недоступны в кэше. Explorer показывает файлы-призраки с накладкой на значке файла. Когда кэш наполнится, элементы-призраки со временем исчезнут. Если пользователь пытается получить доступ к файлу-призраку при наличии

подключения к серверу, файл немедленно копируется в кэш, и накладка призрака на значке файла пропадает.

Когда подкаталог прикрепляется к кэшу службы Автономные файлы, призраки также используются для предоставления пользователю содержимого окружающего пространства имен, не подвергающегося кэшированию. При нахождении вне сети одноуровневые с подкаталогом файлы и каталоги появляются в призрачном состоянии, не давая пользователю повода думать о том, что весь этот другой контекст каким-то образом исчез. Когда файлы и каталоги превращаются в призраки с этой целью, они не кэшируются службой Автономные файлы, а также не доступны для работы вне сети, пока они не будут явным образом прикреплены к кэшу службы Автономные файлы.

Безопасность данных

Целью службы Автономные файлы является предоставление таких же ощущений при доступе к удаленным файлам, которые испытываются пользователем при доступе к локальным файлам. Для достижения этой цели служба Автономные файлы кэширует данные пользователей и их действующий на данный момент доступ к каждому файлу и каталогу в кэше. Эта информация используется драйвером службы Автономные файлы, чтобы добиться соответствующего доступа к объектам в кэше. Зашифрованные файлы, использующие EFS на сервере, также шифруются в кэше.

Служба Автономные файлы кэширует доступ для заданного пользователя как данные, доступные или синхронизируемые от имени этого пользователя. Например, если два пользователя, Able и Baker, совместно пользуются ноутбуком, и пользователь Able пометил файлы на сервере доступными в автономном режиме, файлы копируются в кэш, и кэшируется только доступ пользователя Able. Если внезапно пропадает подключение к серверу, пользователь Baker не сможет получить доступ к файлу в кэше, но когда подключение к серверу возобновится и Baker попытается обратиться к файлу, служба Автономные файлы обновит кэш для отображения доступа пользователя Baker, позволяя обоим пользователям обращаться к файлу при работе вне сети.

Файлы, защищенные с помощью EFS, остаются защищенными, но зашифрованными в контексте безопасности того пользователя, кто первым переносит данные в кэш. При работе вне сети получить доступ к данным в кэше сможет только этот пользователь.

Структура кэша

По умолчанию корневой каталог для кэша службы Автономные файлы расположен в %SystemRoot%\CSC и защищен с помощью DACL, который гарантирует администраторам полный контроль над каталогом, а всем остальным дает доступ по чтению, чтению и запуску и к выводу списка содержимого папки. Как показано на рис. 7.25, ниже корневого каталога находится подкаталог с именем, равным текущей версии схемы базы данных (на данный момент, 2.0.6) и дескриптор безопасности, указывающий SID владельца S-1-5-12, обозначающий, что он принадлежит ограниченному коду и не может быть доступен никем, кроме службы

Автономные файлы. Этот дескриптор безопасности наследуется всеми файлами и подкаталогами ниже каталога версии схемы.

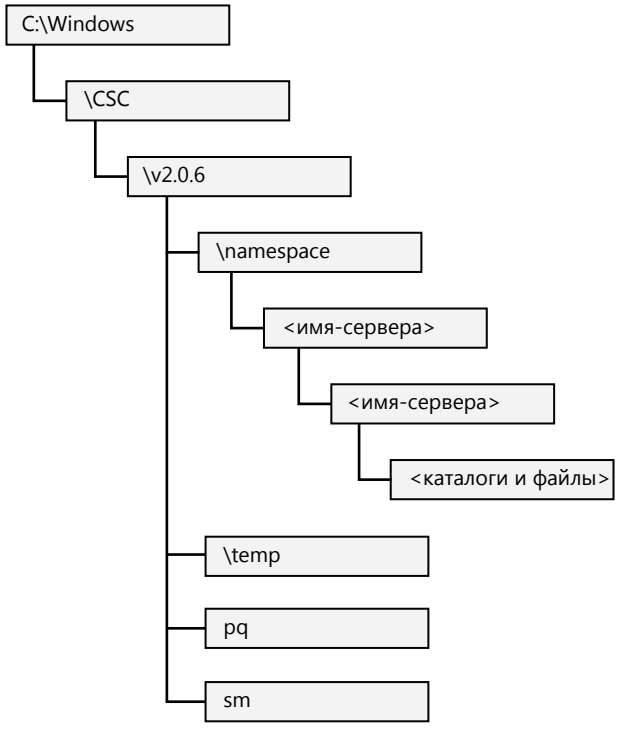


Рис. 7.25. Исходная структура каталогов службы Автономные файлы

В каталоге версии схемы есть два файла и два каталога. Файлы состоят из баз данных очереди по приоритету — Priority Queue (pq), и SID-карты — SID Map (sm). Очередь по приоритету (Priority Queue) является базой данных, в которой отслеживается использование файлов в кэше и их выстраивание в порядке от недавно используемых до наиболее редко используемых. При превышении дисковой квоты кэша потоки агента службы Автономные файлы при выталкивании файлов из кэша переходят от конца очереди в ее начало. Внутри кэша службы Автономные файлы для представления пользователя при сохранении его пользовательского доступа используется внутренний идентификатор пользователя. SID-карта используется для отображения этих внутренних пользовательских идентификаторов на идентификаторы безопасности. Это отображение становится важным при работе без подключения к серверу, и служба Автономные файлы должна сама проверять допустимость пользовательского доступа.

Каталог namespace является корневым для кэша и содержит по каталогу для каждого сервера, кэшируемого службой Автономные файлы. Каталог temp предназначен для шифрования, а также используется как временное место для файлов, убранных из namespace с целью последующего удаления. Каталог temp используется как рабочая область службы Автономные файлы.

Для каждого файла в кэше службы Автономные файлы эта служба добавляет разреженный альтернативный поток данных NTFS по имени CscBitmapStream, в котором содержится битовый массив, показывающий, какие страницы файла были изменены, пока файл был автономным (сервер был недоступен). Каждый бит в битовом массиве представляет внутри файла страницу размером в 4 Кбайта. До появления первой автономной записи в файл этот битовый массив не создается. Записи, увеличивающие размер файла в битовую матрицу не включаются. Если файл, пребывая в автономном состоянии, укорачивается, битовая матрица также укорачивается, чтобы соответствовать новой длине файла. Когда сервер будет доступен в очередной раз, на него будут записаны только измененные страницы.

BranchCache

BranchCache является общим механизмом кэширования содержимого, разработанным для сокращения сетевого трафика, особенно в WAN-сетях. Название BranchCache (кэш филиала) было взято из концепции филиалов компании, подключенных к централизованным серверам этой компании через WAN-каналы, которые обычно в сотни раз медленнее LAN-каналов и кэшируют содержимое, используемое компьютерами, находящимися в этом филиале. Перемещение кэша содержимого в филиал существенно сокращает время доступа к содержимому, поскольку данным не нужно проходить по WAN-каналу.

В отличие от службы Автономные файлы, которая кэширует только файлы, BranchCache кэширует содержимое, то есть все, что может быть идентифицировано с помощью URL-адреса, например файлы, веб-страницы, видеопоток HTTP или даже блоб, доступный из базы данных или из облачной службы.

BranchCache не обращается к файлам в CSC-кэше, поскольку CSC является клиентом BranchCache. Это служба Автономные файлы использует BranchCache для наполнения своего собственного кэша.

BranchCache использует различные протоколы, включая:

- **Блок сообщений сервера — Server Message Block (SMB).** Используется для доступа к файлам на файловых серверах;
- **HTTP(S).** Веб-страницы, видеопотоки и другое содержимое, идентифицируемое с помощью URL;
- **Фоновая интеллектуальная служба передачи — Background Intelligent Transfer Service (BITS).** Используется для передачи файлов и работает через HTTP/TLS 1.1.

Архитектура BranchCache показана на рис. 7.26.

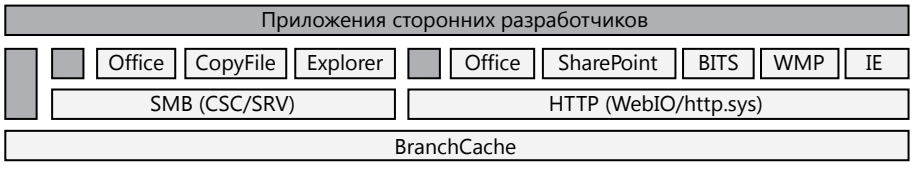


Рис. 7.26. Архитектура BranchCache

На рис. 7.26 показано, что операции BranchCache видны приложениям, обращающимся к кэшируемому содержимому. Когда у клиента включен механизм BranchCache, клиентский запрос к содержимому сервера переносит заголовки и метаданные (конкретный механизм зависит от используемого протокола), оповещая сервер удаленного содержимого, что у клиента включен BranchCache. В таком случае сервер содержимого возвращает информацию о содержимом — content information (CI), — дающую описание этого содержимого, а не само запрошенное содержимое. В CI содержатся хэши всех сегментов и блоков, на которые фрагментировано содержимое. Клиент использует CI для извлечения части или всего содержимого из локального кэша BranchCache. Если какая-то часть содержимого недоступна локально, клиент возвращается к содержимому удаленного сервера для извлечения данных, отсутствующих в локальном BranchCache, и как только данные будут извлечены, предлагает отсутствовавшие данные локальному BranchCache, чтобы такие же данные могли в будущем послужить другим клиентам.

BranchCache работает в двух режимах кэширования, показанных на рис. 7.27:

- **Хост-кэш.** В филиале используется единственный сервер (работающий под Windows Server 2008/R2 или выше) с включенным свойством BranchCache, содержащий весь кэш содержимого для всех систем филиала с включенным BranchCache.

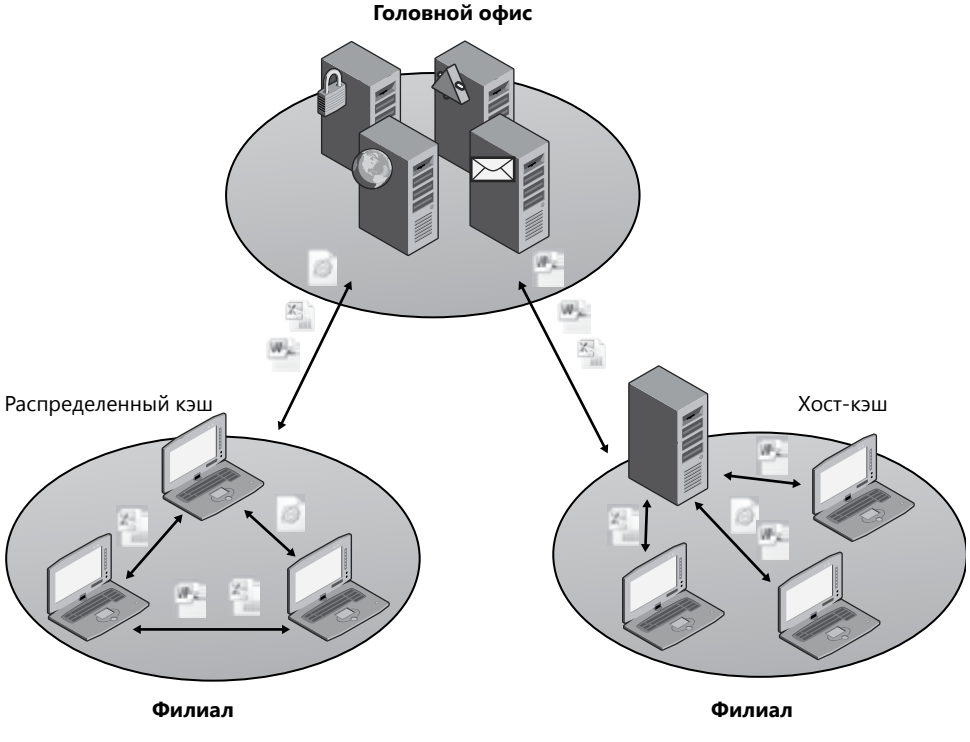


Рис. 7.27. Типы кэширования BranchCache

- **Распределенный кэш.** Вместо сервера хост-кэша, кэширующего содержимое для всего удаленного филиала, клиенты в удаленном филиале кэшируют файлы содержимого самостоятельно. Кэш распределен по всем клиентам одной подсети. Чтобы равномерно распределить содержимое кэша между одноранговыми системами, в филиале не потребуется никаких усилий. В общем, пока не будет занят максимальный объем локального кэша, у каждого клиента имеется копия всего содержимого, к которому он обращался (что приводит к дублированию содержимого в распределенном кэше). Это вполне допустимая ситуация, так как она добавляет кэшу избыточности и гибкости, особенно когда клиенты часто подключаются к сети филиала и отключаются от нее, такое часто происходит, когда пользователи работают на ноутбуках. Распределенный кэш реализуется с использованием одноранговой (пиринговой) сети, в которой для обнаружения с тайм-аутом в 300 мс клиента, у которого в его кэше имеется содержимое, применяется протокол многоадресной передачи Web Services Discovery (WS-D).

BranchCache полностью совместим с оперативным шифрованием (end-to-end encryption), например с IPsec. Точно как же, как и при использовании CSC, для обеспечения работы с кэшированным содержимым используются существующие в Windows механизмы безопасности, чтобы пользователь работал с ним точно так же, как и с некэшированным содержимым.

Механизм BranchCache похож на службу Автономные файлы, но имеет ряд существенных отличий. Наиболее существенное из них состоит в том, что содержимое в BranchCache недоступно, если WAN-сеть отключена. Причина в том, что содержимое идентифицируется создаваемым на сервере и хранящимся на нем хэш-списком, который клиент использует для обнаружения содержимого внутри BranchCache (распределенного или хостированного). У BranchCache имеется ряд особенностей:

- При передаче данных используется AES-шифрование.
- Что касается содержимого, не имеющего файловой основы, то BranchCache кэширует только то содержимое, которое превышает по размеру 64 Кбайт.

(Эти свойства могут быть изменены путем редактирования на сервере параметра реестра HKLM\System\CurrentControlSet\Services\PeerDistKM\Parameters\MinContentLength.)

Режимы кэширования

BranchCache поддерживает два различных локальных кэша на каждой системе с включенным BranchCache (с одной стороны, это могут быть серверы содержимого BranchCache WAN-канала, а с другой стороны, это могут быть клиенты BranchCache и серверы BranchCache хост-кэша):

- *Кэш публикаций*, в котором хранятся метаданные информации о том содержимом, которое опубликовано с использованием API-функций BranchCache Server (PeerDistServer*Xxx*). Структура информации о содержимом содержит хэши различных сегментов и блоков, на которые BranchCache разбивает содержимое при его фрагментации, наряду с секретом, необходимым для

генерации открытых и закрытых идентификаторов содержимого и ключей шифрования.

Публикация обычно считается операцией на стороне сервера, хотя опубликовать содержимое может любой клиент BranchCache. Что касается публикации, BranchCache предлагает два различных подхода своим клиентским приложениям или протоколам для генерирования, управления или сохранения метаданных информации о BranchCache-содержимом:

- Приложение и (или) протокол, использующий ускорение работы с помощью BranchCache, может попросить BranchCache сохранить метаданные информации о содержимом от своего имени (в кэше публикаций BranchCache), позволяя BranchCache управлять временем существования этих метаданных в соответствии с правилами, сроками и лимитами, совместно применяемыми несколькими приложениями, использующими BranchCache. Это достигается публикацией посредством использования API-функций PeerDistServer*Xxx*, и именно так и делается в сборках HTTP-BranchCache и BITS-BranchCache.
- В качестве альтернативы приложение или протокол, желающий воспользоваться ускорением работы за счет использования BranchCache, может попросить BranchCache сгенерировать только метаданные информации о содержимом без их сохранения и просто вернуть метаданные приложению или протоколу. В таком случае приложению или протоколу нужно будет реализовать свой собственный способ хранения этих метаданных или управления ими. Именно этим и занимается объединение SMB-BranchCache.

В обоих случаях протокол, интегрированный с BranchCache, или приложение, использующее BranchCache, напрямую несут ответственность за транспортировку этих метаданных информации о содержимом по WAN-каналу от сервера публикации содержимого к клиентам удаленных филиалов. BranchCache не занимается каналом передачи данных по WAN-соединению и не управляет таким каналом. Транспортировка метаданных информации о содержимом намеренно оставлена за протоколом или приложением, использующим BranchCache-акселерацию, чтобы метаданные могли транспортироваться с тем же уровнем безопасности, аутентификации и авторизации, который бы использовался для извлечения самого содержимого, когда BranchCache не используется. Это согласуется с тем, что с точки зрения безопасности, владение копией информации о содержимом BranchCache для заданного содержимого ничем не отличается от владения всем содержимым, и поэтому для извлечения его копии из других субъектов BranchCache (клиентов, серверов хост-кэша или реализаций сторонних разработчиков) нужна авторизация.

Кэш публикаций не хранит какие-либо реальные данные опубликованного содержимого, в нем хранятся только метаданные информации о содержимом. Публикации, как правило, остаются на довольно продолжительные периоды времени, хотя реальный срок определяется приложением, опубликовавшем содержимое. По умолчанию кэш публикаций ограничен потреблением не более одного процента тома, на котором он находится, что определяется значением, хранящимся в файле %SystemRoot%\ServiceProfiles\NetworkService\

AppData\Local\PeerDistPub. Размер кэша публикаций и то место, где он находится, могут быть изменены с помощью команды NetSh:

- netsh branchcache set publicationcache directory=C:\PublicationCacheFolder
- netsh branchcache set publicationcachesize size=20 percent=TRUE

Кэш переиздания содержит как метаданные (но без секретов), так и сами данные содержимого BranchCache, извлеченного клиентом BranchCache, в виде порций из сегментов и блоков. Все это хранится с целью сделать эти порции содержимого доступными для других клиентов BranchCache. Переизданное содержимое хранится до 28 дней, но может быть выброшено и раньше, если кэш переиздания подойдет к лимиту своего объема и потребуется место для переиздания более свежего содержимого. По умолчанию кэш переиздания не может занимать более пяти процентов тома, на котором он находится, что определяется значением, хранящимся в файле %SystemRoot%\ServiceProfiles\NetworkService\AppData\Local\PeerDistRepub. Размер кэша переиздания и то место, где он находится, могут быть изменены с помощью команды NetSh:

- netsh branchcache set localcache directory=C:\BranchCache\Localcache
- netsh branchcache set localcache size=20 percent=TRUE

BranchCache пытается сохранить кэш переиздания при перезагрузках системы путем использования индексного файла, содержащего базу данных дескрипторов сегментов. Когда система перезагружается, BranchCache контролирует общую целостность кэша переиздания, проверяя, что он был закрыт должным образом. Если кэш переиздания не пройдет эту проверку на целостность, он сбрасывается. Кэш публикаций не сохраняется при перезагрузках. Закрытый кэш публикаций SMB-BranchCache не нуждается в явной сохраняемости, она дается без дополнительных усилий как результат интеграции SMB-BranchCache (рассмотренная ранее) и исходит из того факта, что с использованием SMB все изданное содержимое представлено файлами. Если кэш включен, то перед доступом к файлам осуществляется проверка хэшей.

Настройка

BranchCache можно настроить с помощью редактора локальной групповой политики (рис. 7.28), с помощью сетевой оболочки (NetSh, рис. 7.29) или как часть групповой политики, привносимой администратором (в домене).

Параметры настройки BranchCache:

- Служба реализации BranchCache находится в библиотеке %SystemRoot%\PeerDistSvc.dll. Эта служба запускается при включении BranchCache как на клиентской, так и на серверной стороне, и она взаимодействует с компонентами, работающими в режиме ядра (с драйверами).
- Драйвер расширения HTTP находится в файле %SystemRoot%\System32\Drivers\PeerDistKM.sys. Этот драйвер регистрируется с помощью регистратора сетевых модулей — Network Module Registrar (NMR) — в качестве клиента драйвера http.sys и исследует все HTTP-пакеты, поступающие в систему и отправляющиеся из нее. Он добавляет файлы в кэш и извлекает кэшированную информацию о содержимом для опубликованного содержимого из службы BranchCache, вместо того чтобы отправлять запрос на веб-сервер.

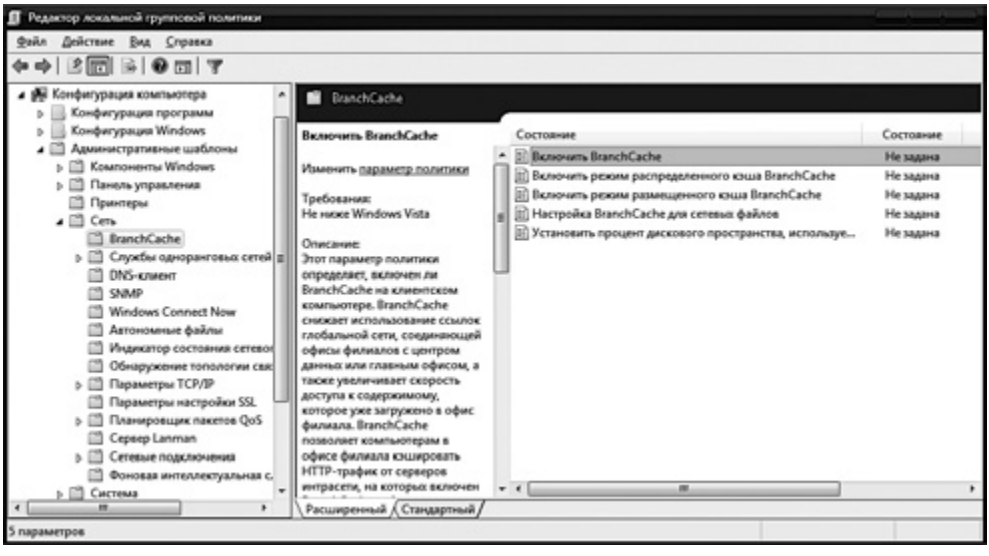


Рис. 7.28. Настройка BranchCache с помощью редактора локальной групповой политики

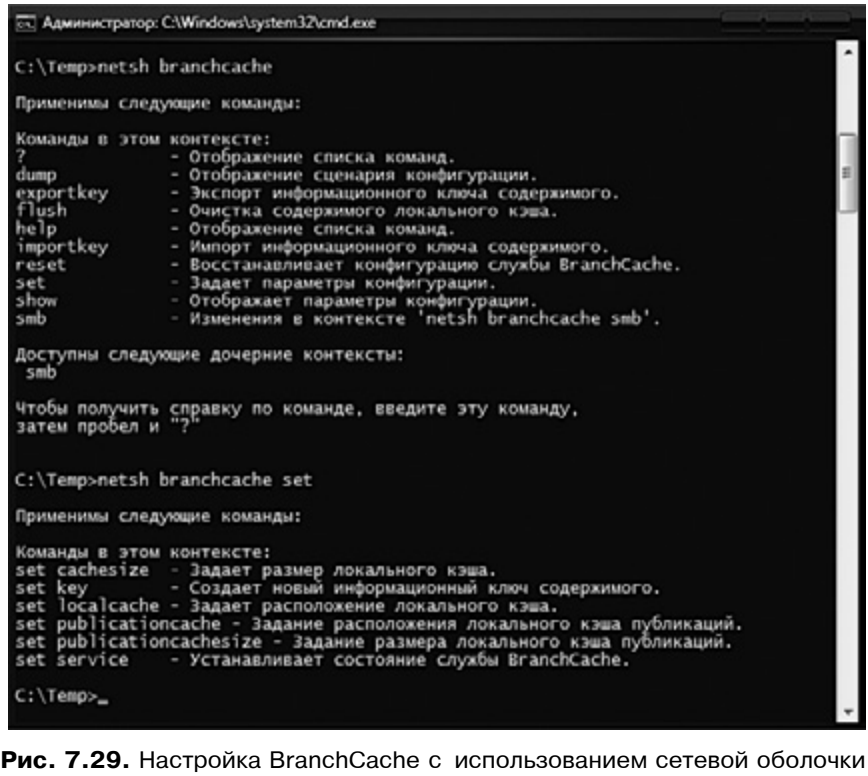


Рис. 7.29. Настройка BranchCache с использованием сетевой оболочки

- FPI-интерфейс BranchCache (PeerDist*Xxx*) экспортируется библиотекой %SystemRoot%\System32\PeerDist.dll, которая использует для связи со службой BranchCache LRPC/ALPC-вызовы.
- HTTP-транспорт BranchCache находится в библиотеке SystemRoot%\System32\PeerDistHttpTrans.dll, реализующей транспорт, поверх которого протокол Кэширования и извлечения коллегиального содержимого: Протокол извлечения — Peer Content Caching and Retrieval: Retrieval Protocol [MS-PCCRR] — ведет обмен данными между клиентами BranchCache и (или) серверами хост-кэша. Каждое сообщение MS-PCCRR заключено в простое транспортное сообщение, которое, в свою очередь, оправляется через HTTP-запрос.
- Поставщик обнаружения веб-служб — Web Services Discovery Provider — находится в библиотеке %SystemRoot%\System32\PeerDistWSDDiscoProv.dll, которая реализует WS-D-протокол для обнаружения того, каким клиентом LAN-сети кэшируется конкретный файл (или часть файла).
- Вспомогательная библиотека сетевой оболочки — BranchCache Network Shell Helper — находится в файле %SystemRoot%\System32\PeerDistSh.dll и является расширением приложения сетевой оболочки — Network Shell (%SystemRoot%\System32\Netsh.exe), — предоставляющей пользователям средства для слежения за службой BranchCache и для настройки этой службы. Вспомогательная DLL-библиотека Network Shell устанавливается путем добавления строкового значения к параметру HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NetSh, которое предоставляет сетевой оболочке Network Shell путь к вспомогательной DLL-библиотеке.
- Автономный вариант всех API-функций BranchCache реализован в библиотеке %SystemRoot%\System32\PeerDistHashPeerDistHash.dll (которая имеется только в системах Windows Server), где содержатся все API-интерфейсы BranchCache и функциональные возможности, и он не требует использования службы BranchCache. Этот компонент разработан для использования другими Windows-свойствами, тесно интегрированными с BranchCache, например механизмом SMB Groveler, генерирующем хэши на сервере.
- Служба гровелера хэша — Hash groveler — находится в файле %SystemRoot%\System32\smbhash.exe (только на системах Windows Server). Гровелер работает на файловом или на веб-сервере, генерирующем хэши, когда клиент запрашивает список хэшей. Гровелер следит за заданным пространством имен или общим ресурсом и обеспечивает обновление BranchCache-хэшей для всех файлов этого пространства имен. Весь ввод-вывод гровелера выполняется с низким приоритетом ввода-вывода, чтобы не мешать нормальной работе системы.

BranchCache использует следующие протоколы, документированные на сайте www.microsoft.com:

- Протокол Кэширования и извлечения коллегиального содержимого: Идентификация содержимого — Peer Content Caching and Retrieval: Content Identification, — как указано в [MS-PCCRC], определяет ранее описанную структуру информации содержимого.
- Протокол Кэширования и извлечения коллегиального содержимого: Протокол обнаружения — Peer Content Caching and Retrieval: Discovery Protocol, — как

указано в [MS-PCCRD], определяет множество адресов (multicast) для служб обнаружения и определения местоположения на основе протокола динамического обнаружения служб — Web Services Dynamic Discovery (WS-Discovery) [WS-Discovery]. У WS-Discovery имеется два режима работы: инициированные клиентом пробники и инициированные сервером уведомления. Все они отправляются через многоадресный IP предопределенной группе. Основная роль в системе кэширования и извлечения содержимого заключается в обнаружении содержимого (Content Discovery).

- Протокол Кэширования и извлечения коллегиального содержимого: Протокол извлечения — Peer Content Caching and Retrieval: Retrieval Protocol, — как указано в [MS-PCCRR], определяет сообщения, необходимые для запроса серверов, играющих коллегиальную роль, или серверов хост-кэша для доступности конкретного содержимого и для извлечения содержимого. Основная роль в системе кэширования и извлечения содержимого заключается в извлечении содержимого (Content Retrieval).
- Протокол Кэширования и извлечения коллегиального содержимого: Протокол хост-кэша — Peer Content Caching and Retrieval: Hosted Cache Protocol, — как указано в [MS-PCHC], определяет основанный на HTTPS механизм для клиентов для уведомления сервера хост-кэша относительно доступности содержимого и для сервера хост-кэша для обозначения интереса к содержимому. Основная роль в системе кэширования и извлечения содержимого заключается в уведомлении о содержимом (Content Notification).
- Протокол Кэширования и извлечения коллегиального содержимого: Расширения протокола передачи гипертекстовых файлов (HTTP) — Peer Content Caching and Retrieval: Hypertext Transfer Protocol (HTTP) Extensions, — как указано в [MS-PCCRTP], определяет шифрование контекста, известное как PeerDist, используемое клиентом HTTP/1.1 и сервером HTTP/1.1 для обмена содержимым друг с другом. Основная роль в системе кэширования и извлечения содержимого заключается в обеспечении зашифрованного обмена содержимым между клиентом и сервером.
- Протокол блока сообщений сервера версии 2.1 — Server Message Block (SMB) Version 2.1 Protocol, — как указано в [MS-SMB2], версия 2.1 этого протокола содержит усовершенствования для обнаружения общих ресурсов с включенным кэшированием содержимого и извлечения метаданных, относящихся к кэшированию содержимого. Основная роль в системе кэширования и извлечения содержимого заключается в извлечении метаданных (хэша) — Metadata (Hash) Retrieval.

Поддержка интеграции SMB-BranchCache требует следующих изменений как клиента, так и сервера. У клиента были расширены функциональные возможности существующих компонентов кэширования на стороне клиента — client-side caching (CSC). У сервера был расширен драйвер SMB Server Driver (srv2.sys) для поддержки хэш-списка, извлекаемого из сервера, и была добавлена новая служба, Служба генерирования хэша SMB — SMB Hash Generation Service (также известная как Гровелер), для управления генерированием, обновлением и удалением хэшей содержимого совместно используемых SMB.

Оптимизированное извлечение данных приложением с помощью BranchCache: SMB-последовательность

В следующей последовательности дается описание того, как содержимое, кэшированное с помощью BranchCache, доставляется приложению, не требуя при этом никаких изменений приложения (рис. 7.30). В этой последовательности рассматривается случай, когда применительно к данному приложению для канала или протокола был сделан выбор в пользу SMB, например приложение открывает файл из удаленного общего ресурса с помощью функции CreateFile (или какой-нибудь функции, вызывающей CreateFile, например fopen) и осуществляет чтение из файла. Если предложение решило извлечь данные посредством HTTP-запроса (при поддержке либо WinHTTP, либо WinInet), последовательность сильно отличается, но все же по-прежнему остается последовательностью, полностью прозрачной для приложения.

BranchCache и SMB объединены через имеющийся в Windows компонент Автономные файлы. Служба Автономные файлы весьма кстати пытается заранее извлечь файлы, связанные через SMB для оптимизации использования сети и улучшения пользовательского восприятия на стороне клиента. Драйвер автономных файлов может временно задержать чтение со стороны приложения, чтобы позволить предварительно извлечь содержимое из BranchCache и иметь возможность оставаться впереди имеющейся в приложении позиции чтения. Эта задержка вычисляется на основе замеренного времени задержки файлового сервера.

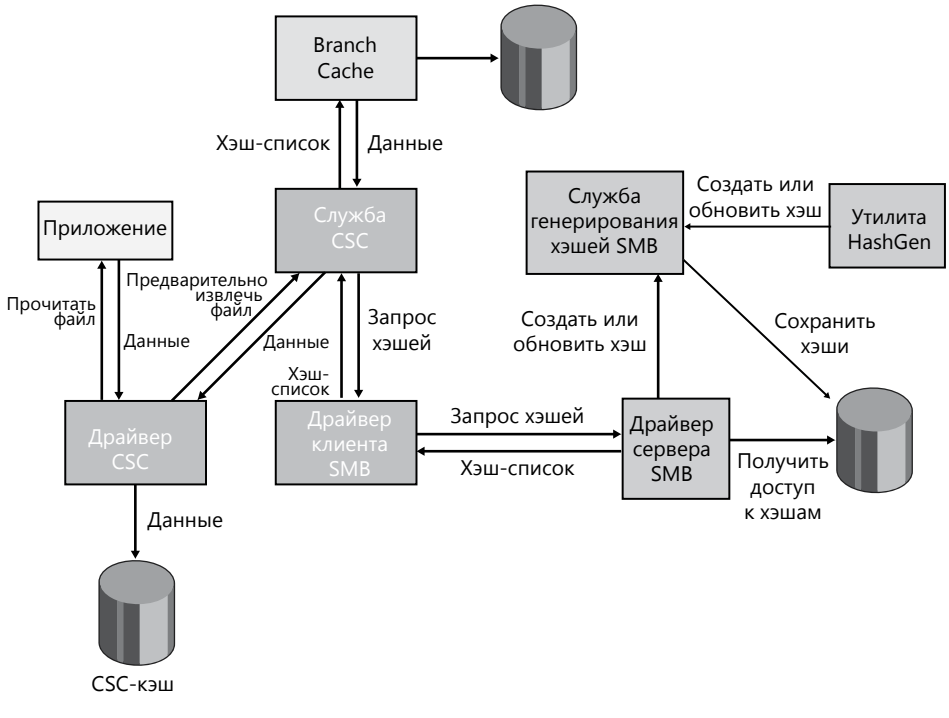


Рис. 7.30. Потоки запросов BranchCache

Извлечение данных начинается с чтением данных приложением из файла на удаленном общем ресурсе SMB. Когда на клиентской стороне разрешена работа службы Автономные файлы, а служба BranchCache не разрешена, запрос приложения на чтение проходит через драйвер автономных файлов на SMB-сервер. Когда на клиентской стороне разрешены обе службы, и Автономные файлы, и BranchCache, происходят следующие действия:

1. Драйвер автономных файлов перехватывает запрос ввода-вывода на чтение и определяет, отвечает ли он следующим специальным условиям для инициирования предварительного извлечения файла:

   Данные еще не хранятся в кэше автономных файлов. Если данные там уже присутствуют, то запрос приложения на чтение будет удовлетворен этими данными без отправки запроса данных на файловый сервер.

   Показатель времени задержки сервера (наблюдавшийся клиентом до сих пор) выше настроенного порогового значения.

   Для совместно используемого файла включено генерирование хэша BranchCache.

   Размер целевого файла составляет не менее 64 Кбайт.

   Чтение зашло за первые 64 Кбайт файла.
2. Если предыдущие условия соблюдены, драйвер автономных файлов уведомляет службу Автономные файлы о необходимости запуска предварительного извлечения файла.
3. Затем служба Автономные файлы извлекает информацию о содержимом из файлового сервера. Если у сервера имеется самая свежая информация о содержимом указанного файла, он возвращает ее клиенту. Если информации о содержимом для указанного файла нет или если она устарела, службе генерирования хэша SMB-протокола на файловом сервере поступает запрос на генерирование новой информации о содержимом для этого файла, и клиенту информация о содержимом возвращена не будет, заставив Службы Автономных файлов пропустить BranchCache-извлечение для этого файла.
4. Если информация о содержимом извлечена из файлового сервера, то служба Автономные файлы использует эту информацию, чтобы попытаться извлечь данные из BranchCache.
5. BranchCache пытается извлечь данные либо из коллегиального кэша, либо из хост-кэша (в зависимости от конфигурации). Если данные найдены, они возвращаются службе Автономные файлы, в противном случае возвращается сообщение об ошибке.
6. Если данные найдены в BranchCache, они записываются в кэш службы Автономные файлы, и поток предварительного извлечения продолжает попытки извлечения данных из BranchCache, пока не извлечет максимум 8 Мбайт данных или не потерпит неудачу в извлечении данных.
7. Когда принадлежащей приложению операции чтения будет разрешено продолжить выполнение, она попытается прочитать данные из кэша службы

Автономные файлы, который заранее заполнен данными из BranchCache, если поток предварительного извлечения успешно извлек данные. В противном случае выполняемой приложением операции чтения будет разрешено обратиться за извлечением данных к серверу. Данные, извлеченные из файлового сервера, затем кэшируются в кэше службы Автономные файлы для последующей публикации в BranchCache.

8. Когда к службе Автономные файлы поступит запрос на предварительное извлечение данных из BranchCache, она также попытается опубликовать любые данные в BranchCache для файла из кэш службы Автономные файлы. Данные файла хранятся в кэш службы Автономные файлы до тех пор, пока кэшу этой службы не понадобится вернуть себе пространство для более свежих файлов. Такие же данные также хранятся в кэше переиздания BranchCache, поэтому они могут совместно использоваться с другими клиентами BranchCache и через различные протоколы или приложения, интегрированные с BranchCache.

Если клиент опять обращается к тому же содержимому (после закрытия файла и его повторного открытия) и содержимое на сервере не изменилось, приложение получит возможность извлечь данные из кэша службы Автономные файлы, не заглядывая в BranchCache. Это называется прозрачным кэшированием.

Если запрошенные данные не могут быть найдены через BranchCache, то как только они будут извлечены из сервера SMB, они будут переизданы для BranchCache, чтобы быть доступными другим клиентам. (Это действие не показано на рис. 7.30.)

Оптимизированное извлечение данных приложением с помощью BranchCache: HTTP-последовательность

В следующей последовательности дается описание того, как содержимое, кэшированное с помощью BranchCache, доставляется приложению, не требуя при этом никаких изменений приложения. В этой последовательности рассматривается случай, когда применительно к данному приложению для канала или протокола был сделан выбор в пользу HTTP, например приложение извлекает содержимое через HTTP-запрос, основанный либо на API WinInet, либо на API WinHTTP.

BranchCache и HTTP тесно интегрированы друг с другом, как с точки зрения HTTP.sys на серверной стороне, так и с точки зрения WinInet и WinHTTP на стороне клиента. В отличие от интеграции SMB-BranchCache, когда механизм BranchCache включен как на клиентской, так и на серверной стороне, исходящие из приложения HTTP-запросы всегда останавливаются, ожидая извлечений из BranchCache. Интеграция HTTP-BranchCache сконцентрирована на минимизации использования WAN-трафика (даже при очень быстрой WAN-сети и очень низкой задержке), и все данные, которые могут быть извлечены из BranchCache, будут переданы через BranchCache.

1. Извлечение данных начинается с выдачи приложением HTTP-запроса.
2. Когда на клиентской стороне разрешена работа BranchCache, клиентский стек HTTP (либо WinInet, либо WinHTTP) добавляет к запросу заголовки, по-

казывающие, что клиент может понимать PeerDist-шифрование HTTP (как это определено в [MS-PCCRTP]).

3. Клиентский стек HTTP отправляет запрос на сервер удаленного содержимого, обычно используя для этого WAN-связь.
4. HTTP-драйвер режима ядра (HTTP.sys) получает запрос на сервер содержимого. Если на этом сервере разрешена работа BranchCache, HTTP.sys направляет копию запроса драйверу HTTP-расширения BranchCache (PeerDistKM.sys), который отслеживает запрос и извлекает информацию о содержимом для данного содержимого (идентифицированного по его URL-адресу и тегам содержимого) из службы BranchCache.
5. Драйвер HTTP режима ядра доставляет HTTP-запрос к связанному с ним веб-серверу, работающему в пользовательском режиме (обычно это IIS или веб-служба) и ждет ответа.
6. HTTP-сервер проводит аутентификацию и авторизацию клиента, генерирует в соответствии с этим ответ и начинает потоковую передачу ответа вниз на HTTP.sys.
7. Поскольку BranchCache включен, HTTP.sys перенаправляет ответ через PeerDistKM.sys.
8. Если информация о содержимом для этого HTTP-содержимого недоступна (или еще не доступна) или если теги содержимого не соответствуют, предпринимаются следующие действия:
   - PeerDistKM.sys отправляет копию потока ответа службе BranchCache для публикации, чтобы следующий запрос на тот же URL-адрес нашел информацию о содержимом.
   - Потоку ответа разрешается вернуться на HTTP.sys в неизменном виде.
   - HTTP.sys отправляет ответ с находящимися в нем фактическими данными и без метаданных BranchCache.
9. Если вместо этого информация о содержимом, касающаяся содержимого HTTP, доступна, и на основе тегов содержимого выясняется, что она соответствует по свежести возвращенному содержимому, предпринимаются следующие действия:
   - PeerDistKM.sys заменяет тело ответа информацией о содержимом, дающей его описание в понятиях BranchCache.
   - Изменяются заголовки ответа, в которые добавляется, что теперь ответ является PeerDist-зашифрованным.
   - Драйверу HTTP.sys возвращается измененный (и, в общем, намного укороченный) поток ответа.
   - HTTP.sys отправляет измененный ответ, содержащий только BranchCache-метаданные информации о содержимом.
10. Клиентская сторона получает ответ. Если ответ содержит информацию о содержимом в формате BranchCache, стек HTTP клиента передает эти метаданные службе BranchCache, и она начинает обслуживать чтение, инициированное первым приложением настоящего содержимого этого ответа, обращаясь

к BranchCache с просьбой извлечь данные содержимого, связанные с размером первого чтения.

11. BranchCache извлекает данные из локального кэша переиздания (если они там доступны) или извлекает расширенный набор, включающий запрошенные данные либо от других BranchCache-клиентов в локальной сети, либо от сервера хост-кэша (в зависимости от конфигурации).
12. Если каких-то из запрошенных данных будет недоставать, BranchCache сигнализирует HTTP-стеку о диапазоне недостающих данных, и HTTP-стек выдает запрос на этот диапазон удаленному серверу для получения недостающих данных (или расширенного набора, включающего эти данные).
13. Как только все данные будут заново собраны для их чтения конкретным приложением, они возвращаются приложению таким способом, который совершенно прозрачен для этого приложения.
14. Последние три действия повторяются до тех пор, пока из рассматриваемого HTTP-ответа не закончится все чтение, инициированное приложением.

Разрешение имен

Разрешением имен называется процесс, благодаря которому текстовые имена, например www.microsoft.com или Mycomputer, преобразуются в числовой адрес, например 192.168.1.1, распознаваемый стеком сетевого протокола. В этом разделе дается описание трех, связанных с TCP/IP протоколов разрешения имен, предоставляемых: система имен домена — Domain Name System (DNS), Windows-служба имен Интернета — Windows Internet Name Service (WINS), и протокол разрешения имен одноранговой сети — Peer Name Resolution Protocol (PNRP).

Система имен домена

Система имен домена — Domain Name System (DNS) — является стандартом (RFC 1035 и др.), согласно которому интернет-имена (такие как www.microsoft.com) преобразуются в соответствующие им IP-адреса. Сетевое приложение, которому требуется получить разрешение DNS-имени в IP-адрес, отправляет поисковый DNS-запрос, используя протокол UDP/IP (TCP/IP используется для запросов, у которых размер ответа превышает 512 байт) на DNS-сервер. Серверы DNS реализуют распределенную базу данных, состоящую из пар имя–IP-адрес, используемых для осуществления преобразования, и каждый сервер обслуживает преобразование для конкретной зоны. Описание подробностей DNS выходит за рамки данной книги, но DNS является основой использования имен в Windows, поэтому данный протокол является основным протоколом разрешения имен, использующимся в Windows.

DNS-сервер Windows реализован в виде службы Windows (%SystemRoot%\System32\Dns.exe), которая включена в серверные версии Windows. Стандартная реализация DNS-сервера основана на текстовом файле, используемом в качестве базы данных, но DNS-сервер Windows может быть настроен на хранение информации о зонах в Active Directory.

Протокол разрешения имен одноранговой сети

Протокол разрешения имен одноранговой сети — Peer Name Resolution Protocol (PNRP) — является распределенным пиринговым протоколом, позволяющим осуществлять динамическое разрешение имен и публикацию исключительно через сети IPv6. Он позволяет устройствам, подключенным к Интернету, публиковать имена пиров (узлов одноранговой сети) и связанных с ними IPv6-адресов, а также публиковать дополнительную информацию. Затем другие устройства будут проводить разрешение имен пиров, извлекая IPv6-адреса и устанавливая соединения.

PNRP предлагает существенные преимущества по сравнению с DNS, благодаря своей распределенной природе — это, по сути, протокол, не использующий сервер (кроме как для самой ранней начальной загрузки), который может потенциально разрастаться вплоть до миллионов имен и который обладает естественной устойчивостью и отсутствием узких мест. Благодаря включению в него служб безопасной публикации имен изменения в записи имен могут осуществляться из любой системы. DNS требует для внесения изменений, как правило, связываться с администратором DNS-сервера. Обновление имен в PNRP также происходит в масштабе реального времени, что подходит для высокомобильных устройств, тогда как DNS кэширует результаты. И наконец, PNRP позволяет не только давать имена компьютерам и службам, разрешая вместе с записями имен публиковать расширенную информацию. Спецификацию протокола разрешения имен одноранговой сети — Peer Name Resolution Protocol [MS-PNRP] — можно найти по адресу www.microsoft.com.

Для приложений и служб PNRP выставляется системой Windows через PNRP API, а также путем расширения функции getaddrinfo Winsock API, рассмотренной ранее в этой главе для выполнения разрешения идентификаторов PNRP ID (рассматриваемых далее), когда адрес включает зарезервированный доменный суффикс .pnrp.net.

Имена одноранговых узлов или пиров PNRP (также называемые идентификаторами P2P ID) состоят из двух компонентов:

- **Полномочия (Authority).** Для безопасных клиентов (записи имен которых подписаны сертифицированными полномочиями) полномочия идентифицируются в виде хэша SHA-1 связанного с ними открытого ключа, а для небезопасных клиентов полномочия имеют нулевое значение. Если клиент безопасен, PNRP проверяет запись имени перед ее публикацией.
- **Классификатор (Classifier).** Классификатор использует простую строку для идентификации службы, предоставляемой пиром, что позволяет на одном и том же устройстве быть предоставленным нескольким службам.

Как показано на рис. 7.31, для создания PNRP ID, PNRP хэширует P2P ID и объединяет его с уникальным 128-разрядным ID, который называется расположением службы (Service location). Расположение службы идентифицирует различные экземпляры одного и того же P2P ID в одном и том же облаке. (PNRP использует два облака: глобальное облако, которое соответствует всем IPv6-адресам Интернета, и облако локальной связи, соответствующее IPv6-адресам с префиксом fe80::/10 и являющееся аналогом подсети IPv4.)

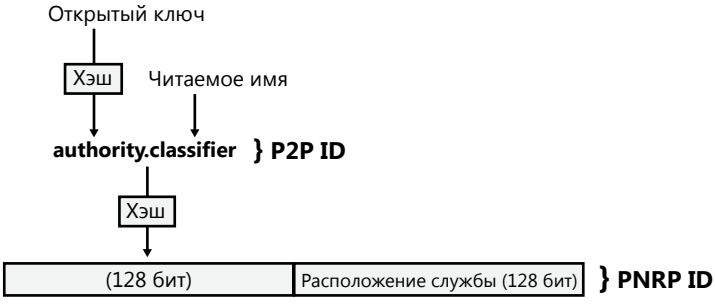


Рис. 7.31. Создание PNRP ID

Разрешение и публикация PNRP

Разрешение имени PNRP происходит в два этапа:

- **Определение конечной точки (Endpoint determination).** На этом этапе запрашивающий пир определяет IPv6-адрес, связанный с пиром, отвечающим за публикацию PNRP ID нужной службы.
- **Разрешение PNRP ID (PNRP ID resolution).** На этом этапе, как только запрашивающий пир обнаружил и подтвердил доступность пира, связанного с IPv6-адресом, он отправляет сообщение с PNRP-запросом для PNRP ID запрашиваемой службы. Пир, предоставляющий службу, отвечает, чтобы подтвердить PNRP ID. Он предоставляет комментарий, составляющий до 4 Кбайт дополнительных данных, например информации о содержимом, связанной со службой.

В ходе выполнения первого этапа PNRP перебирает узлы (пиры) в поиске публикующего узла таким образом, чтобы узел, осуществляющий разрешение имени, мог бы отвечать за контактирующие узлы, которые последовательно ближе к требуемому PNRP ID. Каждый перебор работает следующим образом: как только пир получает сообщение с запросом, он проводит поиск в своем кэше запрошенного PNRP ID. Если будет найдено соответствие, сообщение с запросом направляется именно к нему, в противном случае оно направляется к следующему ближайшему PNRP ID (видя, сколько ID соответствует).

Когда узел получает сообщение с запросом, для которого он не может найти PNRP ID, он проверяет расстояние до любых других идентификаторов в кэше до целевого ID. Если он находит ближайший узел, запрошенный узел отправляет ответ запросившему узлу, этот ответ состоит из IPv6-адреса пира, который ближе всех соответствует целевому PNRP ID. Запрашивающий узел затем может использовать IPv6-адрес для отправки еще одного запроса к узлу с этим адресом. Если ближайших узлов не будет, запрашивающий узел получает уведомление, и этот узел отправляет запрос следующему ближайшему узлу. При условии, что имеются идентификаторы 200, 350, 450, 500 и 800, на рис. 7.32 изображен возможный этап определения конечной точки для примера, в котором пир А пытается найти конечную точку для PNRP 800 (пир Д).

Для публикации своего идентификатора или идентификаторов PNRP ID пир сначала отправляет PNRP-сообщения с публикацией своим ближайшим соседям (по записям в своем кэше, которые имеют идентификаторы, находящиеся на самых низких уровнях), пополнить их кэши. Затем он произвольно выбирает

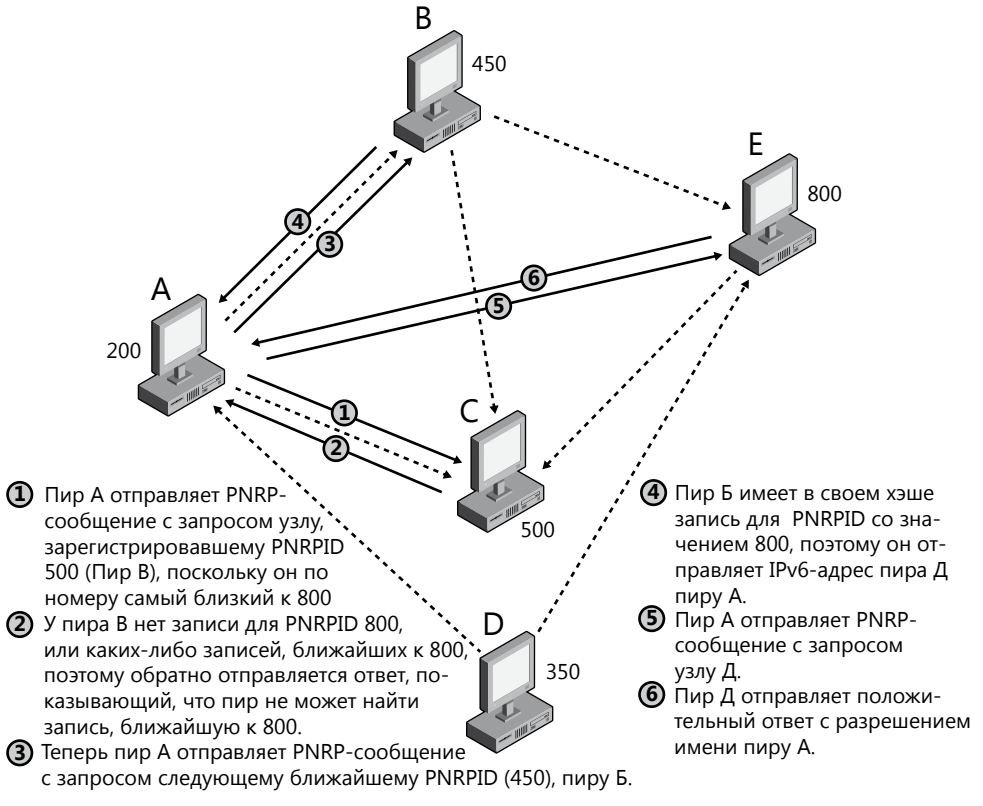


Рис. 7.32. Пример разрешения имени PNRP

узлы в облаке, не являющиеся соседями, и отправляет им PNRP-запросы на разрешение имени для своего собственного PNRP ID. Через ранее описанный механизм этап определения конечной точки рассаживает PNRP ID по кэшам произвольных узлов, выбранных в облаке.

Расположение и топология

Сегодня сетевые компьютеры часто перемещаются между сетями, что требует различных настроек конфигурации, например корпоративной локальной сети и домашней беспроводной сети. Windows включает в себя службу сведений о подключенных сетях — Network Location Awareness (NLA), — позволяющую проводить динамическую конфигурацию и настройку сетевых приложений на основе их расположения, и протокол обнаружения топологии на уровне канала — Link-Layer Topology Discovery (LLTD), — позволяющий проводить настраиваемое обнаружение и отображение сетевых устройств.

Служба сведений о подключенных сетях

Поставщик службы сведений о подключенных сетях — Network Location Awareness (NLA) — реализован в качестве поставщика пространства имен Winsock — Winsock Namespace Provider (NSP) — и предоставляет среду, необ-

ходимую для того, чтобы разрешить перемещаемым по разным сетям компьютерам и устройствам выбирать наиболее подходящие настройки конфигурации. Например, приложение, использующее NLA, может обнаружить, когда пользователь перемещается из высокоскоростной локальной сети на работающую с большими задержками беспроводную сеть и соответствующим образом проводит тонкую настройку использования полосы пропускания. NLA может также определить, когда домашний компьютер, находящийся в локальной сети, может также получить еще одно VPN-подключение к офису, и выбрать соответствующие настройки конфигурации.

Чтобы не ставить разработчиков при определении типа сети и IP-адресов или связанных с ними DNS-имен в зависимость от управляемого вручную сетевого интерфейса, NLA предоставляет API-интерфейс типовых запросов для перечисления всей информации о локальных сетевых соединениях и сопоставляет ее с информацией о сетевом интерфейсе. NLA API также включает уведомления, позволяющие приложениям откликаться на изменения по мере их возникновения. NLA предоставляет приложениям две части информации о расположении:

- **Логический сетевой идентификатор — Logical network identity.** Этот идентификатор основан на логическом сетевом доменном DNS-имени. Если оно отсутствует, NLA использует в качестве идентификатора обычную статичную информацию, сохраненную в реестре наряду с сетевым адресом подсети.
- **Логические сетевые интерфейсы — Logical network interfaces.** Для каждой сети, к которой подключено устройство, NLA создает имя адаптера, идентифицирующее такие интерфейсы, как NIC или RAS-подключения в уникальном виде. Приложения используют имена адаптеров со вспомогательным IP API-интерфейсом (%SystemRoot%\System32\iphlpapi.dll) для запроса информации о интерфейсе и его характеристиках.

Каждая локальная сеть реализована в виде служебного класса со связанным с ним GUID и свойствами. NLA создает экземпляры этого служебного класса при возвращении информации о логической сети. Служебные классы являются схемами, которые описывают пространство имен, они определяют имя, идентификатор и информацию, характерную для пространства имен, общую для всех экземпляров. Эти классы затем используются в сочетании с API-функциями WSALookupService*Xxx* при выполнении разрешения имен.

Самый лучший способ программного получения информации о сетевом размещении заключается в использовании API диспетчера списка сетей — Network List Manager (NLM),\— например INetworkListManager, INetwork, IEnumNetworks, INetworkEvents и т. д.

Индикатор состояния сетевого подключения

Индикатор состояния сетевого подключения — Network Connectivity Status Indicator (NCSI) — определяет в режиме реального времени уровень возможностей сетевого подключения к имеющимся в системе сетевым подключениям. Он загружается службой NLA и работает исключительно как поставщик информации для NLA. Программам, взаимодействующим по сети, NLA позволяет изменять их поведение на основе того, как компьютер подключен к сети. NCSI не

регистрируется в качестве клиента NLA, но он получает непосредственно от NLA конкретные закрытые уведомления. NCSI определяет возможности локальных и интернет-подключений, наличие сетей, доступных через точки доступа, а также возможность и уровень корпоративного подключения.

Уровень возможности подключения к сети оценивается в зависимости от того, имеет или нет система доступ к Интернету и к таким сетевым устройствам, как шлюз по умолчанию, DNS-серверы и веб-прокси серверы. Эта информация о возможности подключения к сети используется различными приложениями, например Networking Tray Icon, Mini Map, Network Connection Wizard, Windows Media Center, DirectAccess, Windows Update и Outlook. Информация NCSI выводится в область уведомлений поверх значка сети. Если пользователь не зарегистрирован, основная часть работы NCSI отключена.

NCSI выполняет основные задачи, рассмотренные в следующих разделах.

Пассивный опрос

Каждые пять секунд (время настраивается в реестре) NCSI активируется для выполнения своей основной работы. Главной целью этой работы является опрос сетевого стека с использованием интерфейса сетевого хранилища — Network Storage Interface (NSI), — и поиск:

1) доказательств получения какого-нибудь трафика. NSI возвращает количество пакетов для каждого сетевого интерфейса. Если интерфейсом были получены какие-нибудь пакеты, у этого интерфейса будет как минимум возможность локального подключения;
2. доказательств того, что этот трафик был получен либо из Интернета, либо из корпоративной сети. Получить его немного сложнее, и такое свидетельство определяется вычислением среднего количества пересылок пакета, прежде чем он покинет системную локальную сеть поставщика интернет-услуг (в домашней или не принадлежащей домену среде) или интранет (в корпоративной среде). NSI возвращает наибольшее количество пересылок, замеченное со времени последнего запроса количества пересылок. Если это количество превышает среднее для заданного IP-семейства (например, IPv4 или IPv6) на заданном интерфейсе, значит, этот интерфейс имеет возможность интернет-подключения;
3) доказательств того, что хост имеет связь с веб-прокси. IP-адреса для веб-прокси серверов будут идентифицированы с помощью механизма автообнаружения веб-прокси — Web Proxy AutoDetect (WPAD), или DNS, и прокси-серверов, настроенных вручную через панель управления интернетом. NSI возвращает подробности текущих TCP-путей из сетевого стека. Если это новый путь к прокси-серверу, значит, этот интерфейс имеет возможность подключения к Интернету;
4) доказательств того, что между системами установлено сопоставление безопасности — Security Association (SA) IPSEC, и того, что хост имеет IPv6-адрес, соответствующий корпоративному сетевому префиксу, определенному в реестре (это пассивное обнаружение возможности корпоративного подключения);

5) доказательств того, что существует достижимый путь, о котором NSI сообщает хосту, с IPv6-префиксом, соответствующим корпоративному сетевому префиксу в реестре. Тогда интерфейс помечается как имеющий возможность корпоративного подключения.

Кроме обработок NSI-запросов, пассивный опрос используется также NCSI для выполнения большинства обработок, связанных со временем. Если подключенных сетей нет, NCSI продолжает опрос, но останавливает его после трех периодов опроса после получения последних данных.

Отслеживание изменений в сетях

NCSI должен знать об изменениях конфигурации интерфейсов в системе. Это обеспечивается двумя отслеживателями событий, которые следят за NSI-уведомлениями об изменениях интерфейса и за уведомлениями об изменениях состояния DHCP.

Когда NCSI обнаруживает, что сеть изменилась, он записывает текущее время в структуру данных, связанную с каждым интерфейсом. Задача пассивного опроса запрашивает эти значения, и если они старше 15 секунд, она выполнит активное зондирование. Задержка в 15 секунд (например, истечение трех периодов опроса) используется для новой оценки состояния возможности подключения к Интернету, если был замечен один недостижимый путь или несколько таких путей.

NCSI регистрируется для DHCP-событий и реагирует на них немедленно (без всякого расхолаживания). Если в этой функции обратного вызова DHCP сообщает, что интерфейс стабилен, активное зондирование для этого интерфейса ставится в очередь.

Отслеживание изменений в реестре

NCSI отслеживает два родительских раздела в реестре на предмет любых изменений в них самих или в их дочерних разделах, используя API уведомлений об изменениях реестра. Любое изменение заставляет NCSI перезагрузить все параметры из каждого раздела:

- HKLM\System\CurrentControlSet\Services\NlaSvc\Parameters\Internet
- HKLM\SOFTWARE\Policies\Microsoft\Windows\NetworkConnectivityStatusIndicator

Активное зондирование

В NCSI есть два механизма для активного тестирования интерфейса с целью обнаружения возможности подключения к Интернету, и оба они могут быть настроены (а также могут быть отключены) с помощью разделов реестра.

При первом активном зондировании интерфейса будет зондироваться Интернет. Это зондирование состоит из попытки загрузить файл http://www.msftncsi.com/ncsi.txt с последующим сравнением содержимого этого файла с ожидаемым значением «Microsoft NCSI». Если сравнение пройдет успешно, активное зондирование считается успешным.

Если NCSI обнаружил прокси-серверы, он проверяет, является ли прозондированный интерфейс лучшим интерфейсом, через который можно достичь первого

прокси-сервера. Если это так, он применяет настройки прокси-сервера к HTTP-запросу. В противном случае он сначала предпринимает попытку без настройки прокси-сервера всего лишь применить эти настройки и предпринимает вторую попытку, если первая оказалась неудачной из-за неудачи с разрешением имени. Это необходимо для поддержки многосетевых сценариев, где один интерфейс подключен через прокси-сервер, а зондируемый интерфейс через него не подключен.

Если активное зондирование пройдет удачно, то в возможность подключения к *Интернету* будет внесено состояние Интернета либо IPv4, либо IPv6. Поскольку NCSI не знает, через какую возможность подключения запрос был удовлетворен, через IPv4 или через IPv6, он строит догадку на основе существования исходных шлюзов для каждого семейства адресов, с выбором IPv4, если точного определения сделать нельзя.

При следующем выполнении активного зондирования, если аппаратный адрес исходного шлюза уже находится в списке известных шлюзов, не использующих прокси-сервер, то вместо веб-зондирования проводится DNS-зондирование. Эта оптимизация приводит к более быстрым результатам. При DNS-зондировании выполняется простой DNS-поиск имени из списка в реестре, по умолчанию используется имя dns.msftncsi.com.

В многосетевых сценариях при обнаружени прокси-сервера поведение HTTP-зондирования изменяется. При выполнении активного зондирования интерфейса проводится проверка, отдано ли сетевым стеком предпочтение этому интерфейсу для достижения первого адреса прокси-сервера. Если это так, веб-зондирование продолжается обычным порядком. Если нет, сначала предпринимается попытка веб-зондирования без использования прокси-сервера. Если она провалится из-за невозможности разрешения имени через DNS, NCSI делает вывод, что он в конечном итоге должен быть за прокси-сервером, и применяет настройки прокси-сервера и проводит повторное зондирование.

Содержимое, извлеченное с помощью HTTP-запроса, сравнивается с известным содержимым, хранящимся в реестре. Если содержимое не соответствует хранящемуся, NCSI предполагает, что интерфейс подключен к сети, доступной через точки доступа (которая перенаправила HTTP-запрос на страницу входа с сеть). Затем используются API-функции диспетчера списка сетей — Network List Manager (NLM) для отправки сообщения объекту служебной оболочки — Shell Service Object (SSO) библиотеки PNIDUI (%SystemRoot%\System32\pnidui.dll), который затем выводит всплывающую подсказку, чтобы показать пользователю, что перед подключением к Интернету ему нужно зарегистрироваться. MAC-адрес шлюза также записывается в известный список шлюзов точки подключения для последующей оптимизации обнаружения прокси-сервера.

NSCI может быть настроен через редактор локальной групповой политики (рис. 7.33) или через реестр.

Обнаружение топологии Link-Layer

Протокол обнаружения топологии Link-Layer — Link-Layer Topology Discovery (LLTD) — работает как по проводным, так и по беспроводным сетям и позволяет приложениям обнаруживать топологию сети. Например, имеющаяся в Windows функция карты сети — Network Map — использует LLTD для создания рисунка

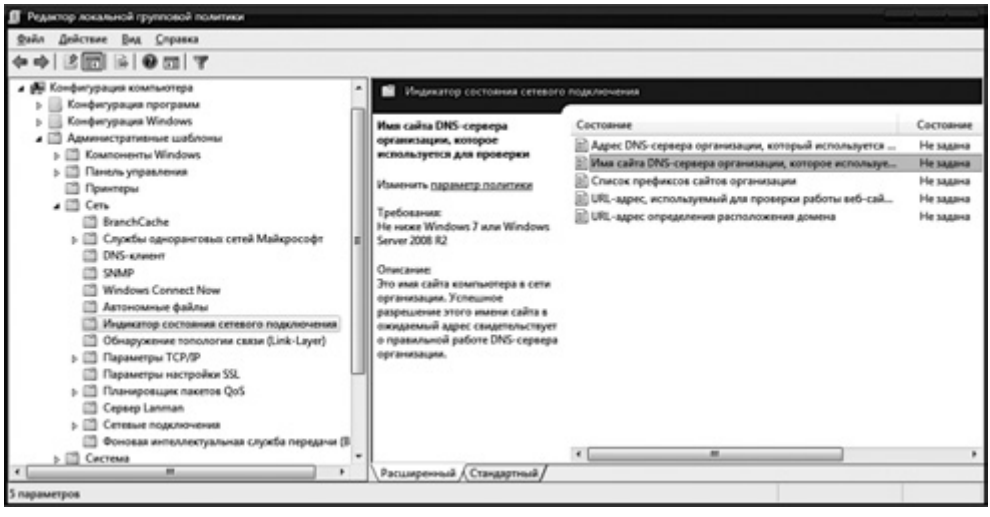


Рис. 7.33. Параметры NCSI в редакторе локальной групповой политики

топологии локальной сети для подключенных устройств, поддерживающих LLTD-протокол. Дополнительно LLTD поддерживает расширения качества обслуживания — Quality of Service (QoS), — что позволяет приложениям диагностировать такие сетевые проблемы, как слабый сигнал беспроводной сети и ограничения пропускной способности домашней сети. Поскольку это расширение работает на канальном уровне взаимодействия открытых систем — OSI, LLTD работает только на единственной локальной сети или на подсети и не может пройти маршрутизаторы, но его возможности вполне подходят для большинства домашних сетей и сетей малых офисов. Спецификация для протокола обнаружения топологии Link-Layer — Link-Layer Topology Discovery protocol [MS-LLTD] — может быть найдена на сайте www.microsoft.com.

Протокол LLTD реализован с использованием таких компонентов, как картограф ввода-вывода LLTD — LLTD Mapper I/O — и LLTD-ответчика — LLTD Responder. Первый из них несет ответственность за процесс обнаружения и за создание сетевых карт. Поскольку им используется протокол, отличный от IP, для непосредственной отправки команд сети через сетевой адаптер LLTD-картограф использует API-функции NDIS. LLTD-ответчик прислушивается к командам обнаружения и отвечает на них информацией о компьютере. Как уже ранее упоминалось, в карте сети показываются только те устройства, у которых есть ответчик.

Драйверы протокола

Сетевые драйверы получают высокоуровневые запросы ввода-вывода и преобразуют их в низкоуровневые запросы сетевого протокола для их передачи по сети. Для выполнения фактического преобразования сетевые API зависят от драйверов транспортного протокола. Отделение API-функций от базовых протоколов дает сетевой архитектуре гибкость, позволяющую каждому API-интерфейсу использовать множество различных протоколов. Бурный рост Интернета и зави-

симость от протокола TCP/IP сделали TCP/IP преимущественным протоколом, используемым в Windows. Агентство перспективных исследований министерства обороны США — Defense Advanced Research Projects Agency (DARPA) — разработало TCP/IP в 1969 году именно в качестве основы для широкомасштабной отказоустойчивой сети, превратившейся в Интернет, поэтому TCP/IP имеет такие подходящие для WAN-сети характеристики, как трассируемость и высокую WAN-производительность. TCP/IP является предпочтительным протоколом Windows, устанавливаемым по умолчанию.

Четырехбайтовые сетевые адреса, используемые протоколом IPv4 на устаревших TCP/IP-стеках, ограничивают количество общедоступных IP-адресов примерно четырьмя миллиардами. Это количество уже почти исчерпано в результате появления все большего количества таких получающих присутствие в Интернете устройств, как мобильные телефоны и наладонные компьютеры. Поэтому признание получает протокол IPv6, имеющий 16-байтовые адреса. Windows включает комбинированный TCP/IP-стек, называемый TCP/IP-стеком следующего поколения — Next Generation TCP/IP Stack, — с одновременной поддержкой как IPv4, так и IPv6, при этом предпочтение отдается протоколу IPv6. При работе в сетях IPv6 стек также поддерживает сосуществование с сетями IPv4 посредством использования туннелирования. TCP/IP-стек следующего поколения предлагает ряд расширенных функций для повышения производительности сети, некоторые из них изложены в приведенном ниже списке:

- **Автонастройка окна получения — Receive Window Auto Tuning.** Протокол TCP определяет *размер окна получения*, который устанавливает объем данных, принимаемый получателем, прежде чем сервер потребует подтверждение. Оптимально размер окна получения должен быть равен произведению пропускной способности на задержку, которое получается при умножении пропускной способности сетевого соединения на его сквозную задержку. Тем самым вычисляется объем данных, которые могут быть «в пути» между отправителем и получателем в любой момент времени. Имеющийся в Windows TCP/IP-стек анализирует условия сетевого соединения и выбирает оптимальный размер окна получения, подстраивая его по мере необходимости при изменении сетевых условий.
- **Составной TCP — Compound TCP (CTCP).** Перегрузка сети происходит при достижении узлом или каналом порога своей пропускной способности. CTCP реализует алгоритм предотвращения перегрузки, отслеживающий пропускную способность сети, время ожидания и потерю пакетов. Он активно увеличивает количество данных, которые могут быть отправлены машиной, когда сеть будет поддерживать это количество, и уменьшает скорость передачи при перегрузке сети. Использование CTCP на широкополосной сети с низким уровнем задержек может существенно увеличить скорость передачи.
- **Явное уведомление о перегрузке — Explicit Congestion Notification (ECN).** Когда TCP-пропадает (его получение не подтверждается), протокол TCP предполагает, что данные были потеряны из-за перегрузки маршрутизатора, и осуществляет управление перегрузкой, которое существенно снижает скорость передачи отправителя. ECN позволяет маршрутизаторам явным образом помечать пакеты, как направленные в условиях перегрузки, и эти метки

читаются имеющимся в Windows TCP/IP-стеке как признак необходимости снижения скорости передачи данных. Снижение скорости передачи приводит к лучшей производительности, чем управление перегрузкой, на основе потерь. По умолчанию ECN-уведомление отключено, поскольку многие устаревшие маршрутизаторы могут сбрасывать пакеты с установленным битом ECN вместо того, чтобы игнорировать этот бит. Для определения, поддерживает ли ваша сеть ECN, можно воспользоваться средством Microsoft Internet Connectivity Evaluation Tool (http://www.microsoft.com/windows/using/tools/igd/default.mspx). Как показано на рис. 7.34, для исследования и изменения характеристик ECN можно воспользоваться сетевой оболочкой (из окна командной строки администратора).

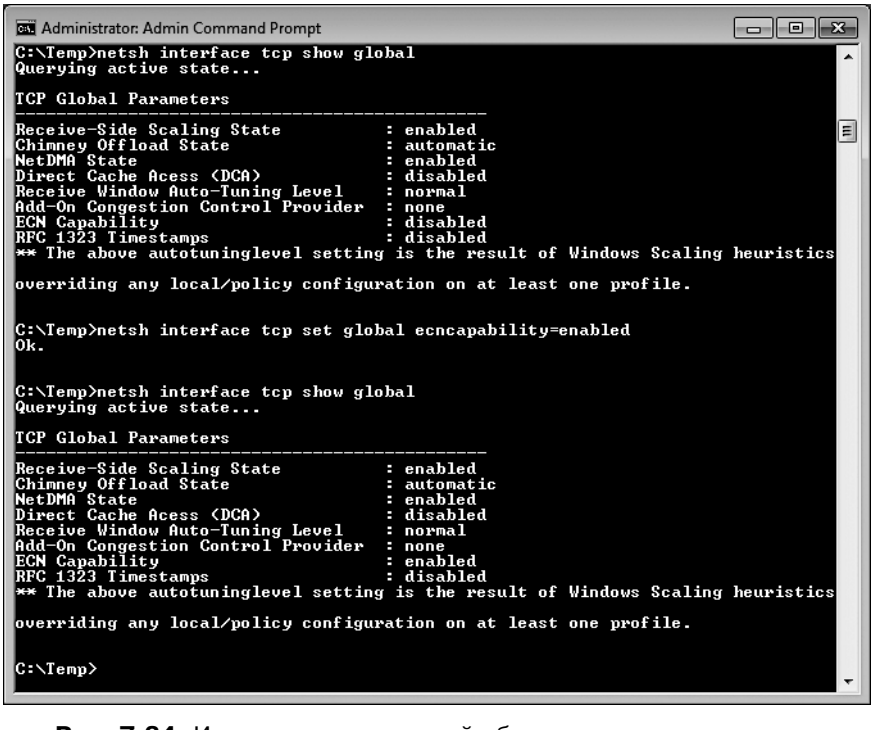


Рис. 7.34. Использование сетевой оболочки для исследования и настройки параметров TCP

- **Улучшение пропускной способности при высоких потерях данных — High-loss throughput improvements**, включая быстрый алгоритм восстановления NewReno — NewReno Fast Recovery Algorithm, алгоритм расширенного селективного подтверждения — Enhanced Selective Acknowledgment (SACK), алгоритм восстановления Forward RTO-Recovery (F-RTO) и алгоритм ограниченного транзита — Limited Transit. Эти алгоритмы сокращают общее количество ретрансляций подтверждений или TCP-сегментов в ходе выполнения сценариев с большими потерями данных, по-прежнему справляясь с целостностью TCP-потока. Это позволяет расширить пропускную способность в таких средах и сохраняет надежную транспортную семантику TCP.

TCP/IP-стек следующего поколения (%SystemRoot%\System32\Drivers\Tcpip.sys), показанный на рис. 7.35, реализует TCP, UDP, IP, ARP, ICMP и IGMP. Для поддержки таких устаревших протоколов, как NetBIOS, которые используют не рекомендованный TDI-интерфейс, сетевой стек также включает в себя компонент под названием TDX (TDI translation — TDI-преобразование), создающий объекты устройств, которые представляют устаревшие протоколы, чтобы клиент мог получить файловый объект, представляющий протокол, и выдать протоколу сетевой ввод-вывод, используя TDI IRP-пакеты. Компонент TDX создает несколько объектов устройств, представляющих различные, доступные клиентам TDI-протоколы: \Device\Tcp6, \Device\Tcp, \Device\Udp6, \Device\Udp, \Device\Rawip и \Device\Tdx.

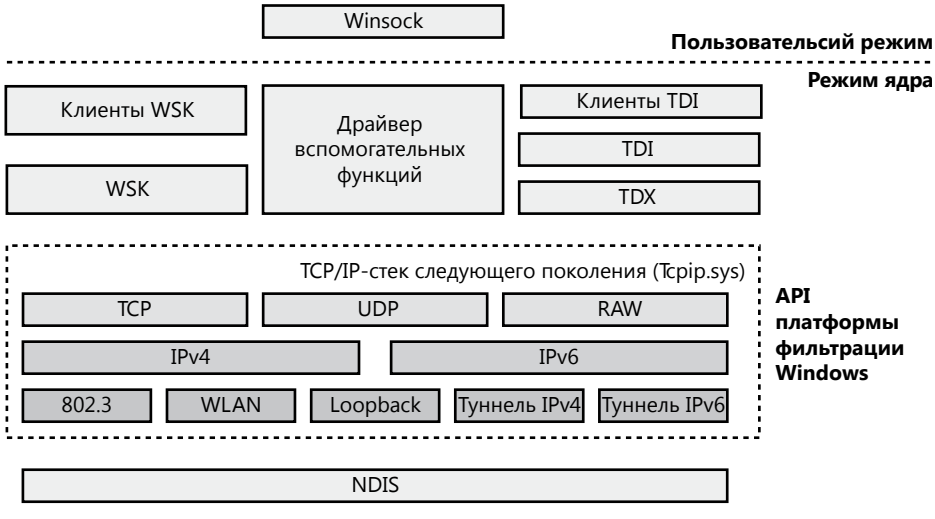


Рис. 7.35. Windows TCP/IP-стек следующего поколения

ЭКСПЕРИМЕНТ: ПРОСМОТР ОБЪЕКТОВ УСТРОЙСТВ TCP/IP

Изучить объекты устройств TCP/IP можно с помощью отладчика ядра при наблюдении за работающей системой. После выполнения команды !drvobj для просмотра адресов каждого принадлежащего драйверу объекта устройства выполните команду !devobj для просмотра имени и других подробностей, касающихся объекта устройства.

```
kd> !drvobj tdx
Driver object (861d9478) is for:
 \Driver\tdx
Driver Extension List: (id , addr)
Device Object list:
861db310  861db440  861d8440  861d03e8
861cd440  861d2318  861d9350
lkd> !devobj 861cd440
Device object (861cd440) is for:
 Tcp6 \Driver\tdx DriverObject 861d9478
```

продолжение ⇩

```
Current Irp 00000000 RefCount 7 Type 00000012 Flags 00000050
Dacl 8b1bc54c DevExt 861cd4f8 DevObjExt 861cd500
ExtensionFlags (0x00000800)
                              Unknown flags 0x00000800
Device queue is not busy.
lkd> !devobj 861db440
Device object (861db440) is for:
 RawIp \Driver\tdx DriverObject 861d9478
Current Irp 00000000 RefCount 0 Type 00000012 Flags 00000050
Dacl 8b1bc54c DevExt 861db4f8 DevObjExt 861db500
ExtensionFlags (0x00000800)
                              Unknown flags 0x00000800
Device queue is not busy.
lkd> !devobj 861d8440
Device object (861d8440) is for:
 Udp6 \Driver\tdx DriverObject 861d9478
Current Irp 00000000 RefCount 0 Type 00000012 Flags 00000050
Dacl 8b1bc54c DevExt 861d84f8 DevObjExt 861d8500
ExtensionFlags (0x00000800)
                              Unknown flags 0x00000800
Device queue is not busy.
lkd> !devobj 861d03e8
Device object (861d03e8) is for:
 Udp \Driver\tdx DriverObject 861d9478
Current Irp 00000000 RefCount 6 Type 00000012 Flags 00000050
Dacl 8b1bc54c DevExt 861d04a0 DevObjExt 861d04a8
ExtensionFlags (0x00000800)
                              Unknown flags 0x00000800
Device queue is not busy.
lkd> !devobj 861cd440
Device object (861cd440) is for:
 Tcp6 \Driver\tdx DriverObject 861d9478
Current Irp 00000000 RefCount 7 Type 00000012 Flags 00000050
Dacl 8b1bc54c DevExt 861cd4f8 DevObjExt 861cd500
ExtensionFlags (0x00000800)
                              Unknown flags 0x00000800
Device queue is not busy.
lkd> !devobj 861d2318
Device object (861d2318) is for:
 Tcp \Driver\tdx DriverObject 861d9478
Current Irp 00000000 RefCount 167 Type 00000012 Flags 00000050
Dacl 8b1bc54c DevExt 861d23d0 DevObjExt 861d23d8
ExtensionFlags (0x00000800)
                              Unknown flags 0x00000800
Device queue is not busy.
lkd> !devobj 861d9350
Device object (861d9350) is for:
 Tdx \Driver\tdx DriverObject 861d9478
```

```
Current Irp 00000000 RefCount 0 Type 00000021 Flags 00000050
Dacl 8b0649a8 DevExt 00000000 DevObjExt 861d9408
ExtensionFlags (0x00000800)
                              Unknown flags 0x00000800
Device queue is not busy.                                    ■
```

Платформа фильтрации Windows Filtering Platform

Windows включает сильную и расширяемую платформу для отслеживания, перехвата и обработки сетевого трафика на всех уровнях сетевого стека. На основе платформы фильтрации Windows Filtering Platform (WFP) работают другие сетевые службы Windows, расширяя базовые сетевые функции драйвера протокола TCP/IP.

В их число входят служба преобразования сетевых адресов — Network Address Translation (NAT), служба IP-фильтрации, служба IP-инспектирования и служба безопасности IP-протокола — Internet Protocol Security (IPsec). На рис. 7.36 показана интеграция различных компонентов WFP с TCP/IP-стекам. Сюда включены:

- **Механизм фильтрации (Filter engine).** Механизм фильтрации реализован как в пользовательском режиме, так и в режиме ядра и выполняет все фильтрующие операции в отношении сети. Каждый компонент механизма фильтрации состоит из уровней фильтрации, по одному для каждого компонента сетевого стека. Механизм пользовательского режима, отвечающий за RPC и ключевую политику IPsec, кроме всего прочего, содержит примерно 10 фильтров, а механизм режима ядра, выполняющий фильтрацию сетевого и транспортного уровней TCP/IP-стека, содержит около 50 фильтров.
- **Прокладки (Shims).** Прокладки относятся к компонентам режима ядра и находятся между сетевым стеком и механизмом фильтрации. Они отвечают за принятие решения на разрешение или блокировку сетевого трафика на основе своего фильтрующего поведения, определяемого механизмом фильтрации. Прокладка работает в три этапа: она проводит разбор входящих данных, чтобы сравнить входящие значения с записями в механизме фильтрации, вызывает механизм фильтрации, чтобы вернуть действие, основанное на входящих значениях, а затем интерпретирует действие (например, сбрасывает пакет).
- **Основной механизм фильтрации — Base filtering engine (BFE).** BFE является службой пользовательского режима (%SystemRoot%\System32\Bfe.dll), которая управляет всеми WFP-операциями. Эта служба отвечает за добавление и удаление фильтров из WFP-стека, управляя настройкой фильтров и обеспечивая безопасность базы данных фильтров.
- **Драйверы внешних вызовов — Callout drivers.** Драйверы внешних вызовов являются компонентами режима ядра, добавляющими специальную фильтрацию, выходящую за пределы основной поддержки, предоставляемой WFP. Драйверы внешних вызовов связывают функции внешнего вызова с одним или несколькими уровнями фильтрации режима ядра, и WFP разрешает выполнять функциям внешнего вызова глубокую инспекцию и модификацию пакета. Например, в виде драйверов внешних вызовов реализованы служба преобразования сетевых адресов — Network Address Translation — и служба IPsec.

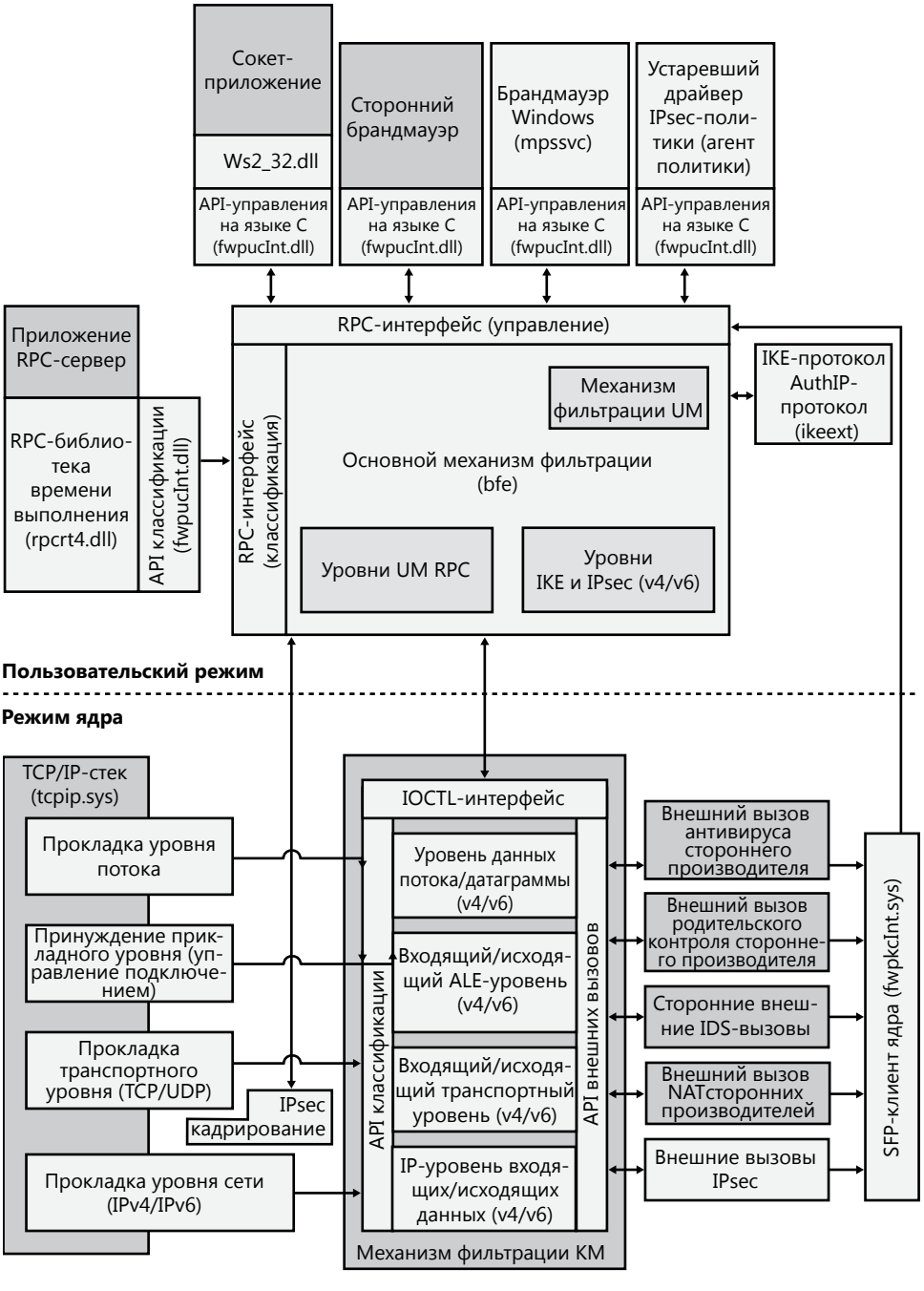


Рис. 7.36. Архитектура платформы фильтрации Windows

Преобразование сетевых адресов

Преобразование сетевых адресов — Network Address Translation (NAT) — является службой маршрутизации, позволяющей нескольким закрытым IP-адресам отображаться на один открытый IP-адрес. Без NAT каждому компьютеру локальной сети пришлось бы назначать для связи по Интернету открытый IP-адрес. NAT позволяет назначать IP-адрес одному компьютеру локальной сети, а другим компьютерам использовать закрытые IP-адреса и быть подключенными к Интернету через этот первый компьютер. NAT осуществляет преобразования между закрытыми IP-адресами и открытым IP-адресом по мере необходимости, прокладывая маршруты для пакетов между компьютерами локальной сети и Интернетом.

NAT-компоненты Windows состоят из драйвера устройства NAT, %SystemRoot%\System32\Drivers\ipnat.sys, который служит интерфейсом со стеком WFP в качестве драйвера внешнего вызова, а также редактором пакетов, который может выполнять дополнительную обработку пакетов, выходящую за рамки преобразования адресов и портов.

IP-фильтрация

Windows включает очень простую IP-фильтрацию, которую пользователь может выбрать для того, чтобы разрешить служить входами и выходами сети только определенным портам или IP-протоколам. Хотя этой возможностью можно воспользоваться для защиты компьютера от несанкционированного доступ к сети, ее недостатком является статичность и невозможность автоматического создания новых фильтров для трафика, инициированного приложениями, запущенными на компьютерах.

Windows также включает возможность установки межсетевой защиты на главном компьютере, которая называется брандмауэром Windows и выходит за рамки только что рассмотренной базовой фильтрации. Чтобы обеспечить динамическую межсетевую защиту, которая отслеживает транспортный поток с целью отличить TCP/IP-трафик, исходящий от локальной сети, и незатребованный трафик, исходящий от Интернета, брандмауэр Windows использует WFP. Когда в интерфейсе включен брандмауэр Windows, можно применить один из трех профилей — общий (public), частный (private) и профиль домена (domain). По умолчанию, когда выбран общий профиль (или пока профиль не выбран), весь предоставленный без запроса входящий трафик, получаемый компьютером, игнорируется. Пользователь или приложение может определить исключения, чтобы службы, запущенные на компьютере, например общий файл, или принтер, или веб-сайт, могли быть доступны из других компьютеров.

Служба Брандмауэр Windows, выполняемая в процессе Svchost, использует BFE для передачи правил исключений, определенных в пользовательском интерфейсе настройки, драйверу IPNat. Механизм фильтрации WFP выполняет функции обратного вызова каждого зарегистрированного драйвера внешнего вызова по мере его обработки как входящих, так и исходящих IP-пакетов. Функция обратного вызова может предоставлять функциональные возможности NAT путем изменения в пакете адресов отправителя и получателя или в качестве брандмауэра за счет возвращения TCP/IP кода состояния, который требует, чтобы TCP/IP сбросил пакет и прекратил его обработку. В режиме ядра

брандмауэр Windows использует драйвер защитной службы Microsoft Protection Service (%SystemRoot%\System32\Drivers\Mpsdrv.sys), который предоставляет поддержку для PPTP- и FTP-фильтрации, поскольку эти протоколы предоставляют свои собственные независимые каналы управления и данных. Драйвер должен анализировать канал управления, чтобы определить, каким каналом данных нужно манипулировать. Драйвер также используется для отображения окон уведомлений, когда приложение начинает прослушивать сокет.

Служба безопасности протокола Интернета

Служба безопасности протокола Интернета — Internet Protocol Security (IPsec), — которая объединена с Windows TCP/IP-стеком, помогает защищать одноадресные[1] IP-данные от таких атак, как подслушивание (eavesdropping), сниффер-атаки (sniffer attacks), изменение данных, имитация соединения по IP-адресу (IP address spoofing) и атак типа man-in-the-middle («человек в середине»), если идентичность удаленной машины может быть проверена, например, как в VPN.

IPsec можно использовать для предоставления глубоко эшелонированной обороны (defense-in-depth) против сетевых атак с не внушающих доверия компьютеров, конкретных атак, которые могут привести к отказам в обслуживании со стороны приложений, служб или сети, к повреждению данных, к краже данных и к краже учетных данных пользователей, к административному контролю над серверами, другими компьютерами в сети. IPsec помогает обороняться против сетевых атак посредством криптографических служб безопасности, безопасных протоколов и динамического распределения ключей.

Для одноадресных IP-пакетов, отправляемых между надежными узлами сети, IPsec предоставляет следующие свойства:

- аутентификацию источника данных, в целях которой проверяется источник IP-пакета и гарантируется, что не прошедшие аутентификацию стороны не могут получить доступ к данным;
- целостность данных, в целях которой IP-пакет защищается от незаметного внесения изменений в ходе его перемещения;
- конфиденциальность данных, в целях которой перед передачей шифруется полезная составляющая IP-пакетов. Конфиденциальность данных гарантирует, что прочитать и интерпретировать содержимое пакетов сможет только IPsec-узел того же уровня, с которым связан компьютер. Это свойство является дополнительным;
- антиповтор (или защиту от повтора), гарантирующий уникальность и невозможность повторного использования каждого IP-пакета. Это свойство не дает атакующим перехватывать IP-пакеты и вставлять измененные пакеты в поток данных между компьютером-источником и компьютером-приемником. При использовании антиповтора атакующие не могут ответить на перехваченные сообщения для установки сеанса или для получения несанкционированного доступа к данным.

IPsec можно использовать для защиты от сетевых атак путем настройки фильтрации IPsec-пакетов узла и принудительного включения надежной связи.

[1] Сама IPsec поддерживает и многоадресные данные, но их не поддерживает Windows-реализация.

При использовании IPsec для фильтрации IPsec-пакетов узла IPsec может разрешить или заблокировать определенные типы одноадресного IP-трафика на основе комбинации адресов источника и приемника и определенных протоколов и определенных портов.

В среде Active Directory для настройки доменов, сайтов и подразделений организации могут использоваться групповые политики, а IPsec-политики (называемые правилами безопасности соединения) могут затем назначаться в качестве требуемых для объектов групповых политик — Group Policy objects (GPO) — через настройки конфигурации Брандмауэра Windows в режиме повышенной безопасности (Advanced Security). Кроме того, можно настраивать и назначать локальные IPsec-политики. Правила безопасности соединения, основанного на Active Directory, хранятся в Active Directory, а копия текущей политики сохраняется в кэше локального реестра. Правила безопасности локального соединения хранятся в реестре локальной системы.

Для установки надежных соединений IPsec использует взаимную аутентификацию и поддерживает следующие методы аутентификации через AuthIP, Microsoft-расширение протокола обмена ключами в Интернете — Internet Key Exchange (IKE):

- интерактивные пользовательские учетные данные Kerberos 5 или интерактивные пользовательские учетные данные NTLMv2;
- пользовательские сертификаты x.509;
- SSL-сертификаты компьютера;
- сертификаты состояния NAP;
- анонимную аутентификацию (необязательную аутентификацию);
- предварительный ключ.

Если расширение AuthIP недоступно, IPsec поддерживает также простой IKE. Windows-реализация IPsec базируется на IPsec-приглашениях для обсуждений — Requests for Comments (RFC). Windows IPsec-архитектура включает Брандмауэр Windows в режиме повышенной безопасности, устаревшего агента политики IPsec (IPsec Policy Agent), IKE и протоколы интернет-аутентификации — Authenticated Internet Protocol (AuthIP), — а также драйвер внешнего вызова IPsec WFP, описание которого дается в следующем списке:

- **Брандмауэр Windows в режиме повышенной безопасности.** В дополнение к ранее рассмотренным функциям фильтрации, служба Брандмауэр Windows также отвечает за предоставление настроек конфигурации защиты и политики для службы IPsec, которая может быть настроена через групповые политики либо локально, либо в домене Active Directory.
- **Устаревший агент политики IPsec.** Этот агент запускается в виде службы. В оснастке Службы (Services) консоли управления Microsoft — Microsoft Management Console (MMC), Агент политики IPsec появляется в списке компьютерных служб под таким же именем. Этот агент получает устаревшую IPsec-политику от домена Active Directory или от локального реестра, а затем передает фильтры IP-адресов драйверу и аутентификации IPsec, а также передает настройки безопасности протоколу IKE. Эти политики принимают

на себя включение совместимости с устаревшими версиями Windows, которые реализуют управление IPsec через Active Directory.

- **IKE и AuthIP.** IKE является протоколом, поддерживающим службы аутентификации и согласования ключей, требуемые IPsec. Для исходящего трафика IKE ждет запросов для согласования сопоставлений безопасности — security associations (SA) от драйвера IPsec, согласовывает сопоставления безопасности, а затем отправляет SA-настройки обратно драйверу IPsec. Для входящего трафика IKE получает запрос на согласование непосредственно от удаленного одноуровневого узла, и весь остальной трафик от этого узла сбрасывается, пока не будут согласованы сопоставления безопасности. SA-сопоставления являются сочетанием взаимно согласованных настроек политики IPsec и ключей, которые определяют службы безопасности, механизмы и ключи, использующиеся для помощи в организации безопасных связей между одноуровневыми узлами IPsec. Каждое SA-сопоставление является однонаправленным или симплексным соединением, защищающим переносимый трафик. По запросу IPsec-драйвера IKE проводит согласование основного и быстрого режима SA. IKE-согласование защищается основным режимом IKE SA — IKE main mode SA (или ISAKMP). Трафик приложений защищается быстрым режимом (или IPsec) SA. AuthIP является частным расширением IKE, поддерживаемым Windows Vista и более поздними версиями, а в Windows 7 и в Windows Server 2008 R2 также добавлена поддержка IKEv2, равного стандартизованного расширения. Она дополнена вторичным механизмом аутентификации для повышения безопасности и упрощения поддержки и настройки IPsec.
- **Драйвер внешнего вызова IPsec WFP.** Этот драйвер является драйвером устройства (%SystemRoot%\System32\Drivers\Fwpkclnt.sys), который привязан к WFP и обрабатывает пакеты, передаваемые через драйвер TCP/IP. Драйвер IPsec отслеживает и защищает исходящий одноадресный IP-трафик, а также он отслеживает, расшифровывает и проверяет исходящие одноадресные IP-пакеты. WFP получает фильтры из агента политики IPsec и инициирует внешний вызов функции, которая затем разрешает, блокирует или защищает пакеты в соответствии с требованиями. Для защиты трафика IPsec-драйвер использует действующие настройки SA-сопоставлений или требует создания новых SA-сопоставлений.

Для создания правил безопасности соединений и управления ими можно воспользоваться Мастером создания правила для нового безопасного подключения (New Connection Security Rule Wizard), показанным на рис. 7.37, который входит в состав MMC-оснастки Брандмауэр Windows в режиме повышенной безопасности — Windows Firewall with Advanced Security (%SystemRoot%\System32\Wf.msc). Эта оснастка может использоваться для создания, изменения и сохранения правил безопасности локального подключения или правил безопасности подключения на основе Active Directory. Кроме этого для управления правами безопасности можно воспользоваться утилитой Netsh с командой netsh advfirewall consec. После установки IPsec-защищенного соединения, используя оснастку Брандмауэр Windows в режиме повышенной безопасности или утилиту Netsh с командой netsh advfirewall monitor, можно отслеживать информацию IPsec как для локальных, так и для удаленных компьютеров.

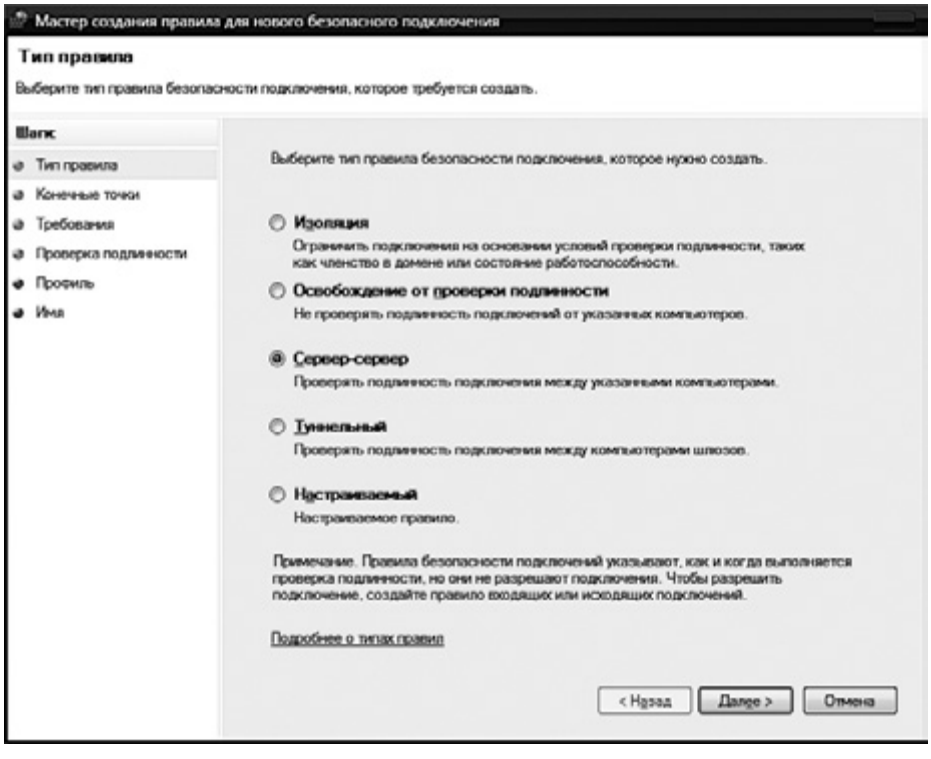


Рис. 7.37. Мастер создания правила для нового безопасного подключения

NDIS-драйверы

Когда протокольный драйвер хочет прочитать или записать сообщения, имеющие его протокольный формат из сети или в сеть, драйвер должен делать это, используя сетевой адаптер. Ожидать от протокольных драйверов понимания нюансов каждого имеющегося на рынке сетевого адаптера было бы неразумно (количество специализированных сетевых адаптеров исчисляется тысячами), поэтому поставщики сетевых адаптеров предоставляют драйверы устройств, способные брать сетевые сообщения и передавать их через специализированное оборудование поставщика. В 1989 году компании Microsoft и 3Com совместно разработали спецификацию интерфейса сетевых драйверов — Network Driver Interface Specification (NDIS), — позволяющую протокольным драйверам связываться с драйверами сетевых адаптеров без учета особенностей тех или иных устройств. Драйверы сетевых адаптеров, соответствующие NDIS, называются *NDIS-драйверами* или драйверами *мини-портов NDIS*. С Windows 7 и Windows Server 2008 R2 поставляется версия NDIS 6.20.

Библиотека NDIS (%SystemRoot%\System32\Drivers\Ndis.sys) проводит границу, существующую между сетевыми транспортами, например драйвером TCP/IP и драйверами адаптеров. Библиотека NDIS является вспомогательной библиотекой, которую клиенты NDIS-драйвера используют для форматирования команд, отправляемых ими NDIS-драйверам. NDIS-драйверы работают с библио-

текой, чтобы получать запросы и отправлять ответы. Взаимоотношения между различными компонентами, связанными с NDIS, показаны на рис. 7.38.

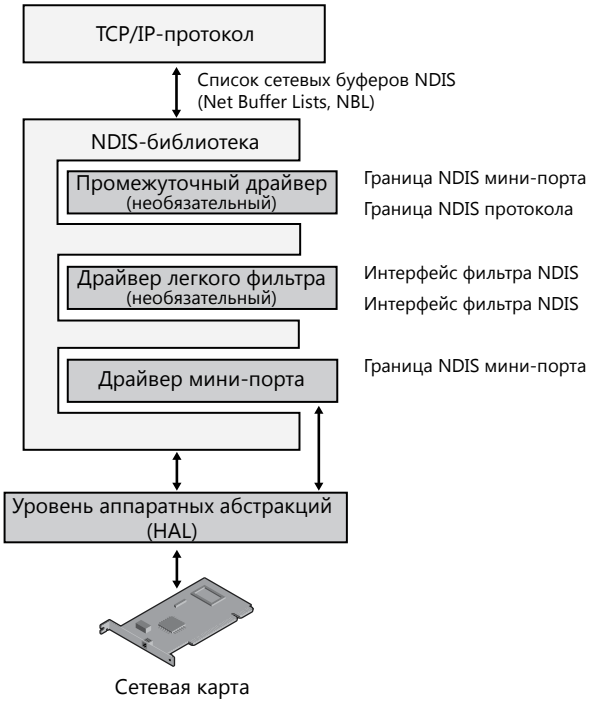


Рис. 7.38. NDIS-компоненты

Вместо простого предоставления вспомогательных процедур NDIS-границы, библиотека NDIS предоставляет NDIS-драйверы со всей средой выполнения. NDIS-драйверы не следуют стандарту модели драйверов устройств ввода-вывода Windows, и они не могут работать без инкапсуляции, предоставляемой им библиотекой NDIS. Этот изоляционный уровень помещает NDIS-драйверы в оболочку настолько основательно, что NDIS-драйверы не могут принимать и обрабатывать IRP-пакеты. Точнее говоря, протокольные драйверы, например TCP/IP, вызывают функцию в NDIS-библиотеке, NdisAllocateNetBufferList, и передают пакеты минипорту NDIS, вызывая функцию в NDIS-библиотеке (NdisSendNetBufferLists). Кроме того, для упрощения разработки все компоненты TCP/IP-стека Windows следующего поколения используют структуру NET_BUFFER_LIST, включая TCP/IP и WSK, что упрощает связь с помощью NDIS.

NDIS обладает следующими свойствами:

- Драйверы NDIS могут сообщить, активна или нет их сетевая среда, что позволяет Windows показывать в панели задач значок подключенной или отключенной сети. Это свойство также позволяет протоколам и другим приложениям знать об этом состоянии и выдавать соответствующую реакцию. Например, TCP/IP-транспорт использует эту информацию для определения необходимости перевычисления адресной информации, полученной от DHCP.

- Работа драйверов NDIS может быть остановлена и продолжена, что позволяет перенастроить их на работающем компьютере, например добавив или удалив драйвер легкого фильтра NDIS (NDIS Lightweight Filter). Легкий фильтр заменяет большинство экземпляров промежуточных драйверов NDIS, которые использовались до выхода NDIS версии 6. (Промежуточные драйверы попрежнему поддерживаются в NDIS 6, но из-за своей сложности они подходят только для решения небольшого класса задач.) Более подробно драйверы легких фильтров будут рассмотрены в следующих разделах.
- Разгрузка TCP/IP, включая разгрузку задач и Chimney-разгрузку. Разгрузка задач позволяет карте сетевого интерфейса реализовать некоторую часть или весь стек протокола TCP/IP, обеспечивая тем самым существенное повышение сетевой производительности. NDIS включает поддержку разгрузки задач IPsec версии 2 (IPsec Task Offload Version 2), в которую включена поддержка дополнительных криптографических пакетов, используемых в IPsec, например поддержка AES, а также IPv6. Chimney-разгрузка предоставляет непосредственное соединение (так называемое chimney) между сетевыми приложениями и аппаратурой сетевой карты, позволяя этой карте выполнить еще большую разгрузку и управление состоянием сетевого подключения. Эти операции разгрузки могут повысить производительность системы за счет освобождения центрального процессора от выполнения таких задач.
- Масштабирование на принимающей стороне позволяет системам с несколькими процессами выполнять операции получения пакетов на основе более эффективного использования доступных целевых процессоров. NDIS поддерживает интерфейс масштабирования на стороне получателя — receive-side scaling (RSS) interface — на аппаратном уровне и целевые прерывания и DPC-вызовы, направленные на соответствующие процессоры.
- Пробуждение по локальной сети (wake-on-LAN) позволяет сетевым адаптерам, имеющим такую возможность, выводить систему из ожидающего состояния электропитания. События, способные переключить сетевой адаптер на отправку сигнала системе, включают в себя подключения носителей (например, присоединение сетевого кабеля к адаптеру), получение характерных для протокола набора сигналов (TCP/IP-транспорт просит для пробуждения запросы протокола преобразования адресов — Address Resolution Protocol [ARP]), зарегистрированного протоколом, и, для Ethernet-адаптеров, получение магического пакета (сетевого пакета, содержащего 16 последовательных копий адреса адаптера Ethernet).
- Отделение заголовка от данных позволяет совместимым сетевым картам повысить сетевую производительность за счет разделения частей Ethernet-фрейма, относящихся к заголовку и к данным на разные буферы с последующим размещением этих буферов в меньших по размеру участках памяти, чем тот, который потребовался бы в случае объединения буферов. Это позволяет добиться более эффективного использования памяти, а также лучшего кэширования, потому что несколько заголовков могут поместиться на одной странице.
- Спецификация NDIS, ориентированная на соединения — Connection-oriented NDIS (CoNDIS), — позволяет NDIS-драйверам управлять ориентирован-

ной на соединения средой (как правило, WAN), например ISDN- или PPP-устройствами.

Интерфейсы, предоставляемые NDIS-библиотекой NDIS-драйверам для взаимодействия с аппаратурой сетевого адаптера, доступны через функции, которые непосредственно преобразуются в соответствующие функции в HAL.

ЭКСПЕРИМЕНТ: ВЫВОД СПИСКА ЗАГРУЖЕННЫХ NDIS-МИНИ-ПОРТОВ

Библиотека расширения отладчика ядра Ndiskd включает команды !miniports и !miniport, которые позволяют вывести список загруженных мини-портов, используя отладчик ядра и, задавая адрес блока мини-порта (структуры данных, которую Windows использует для отслеживания мини-портов), просматривать подробную информацию о драйвере мини-порта. В следующем примере команды !miniports и !miniport показаны использующимися для вывода списка всех мини-портов а затем для вывода особенностей, касающихся мини-порта, ответственного за создание интерфейса между системой и адаптером PCI Ethernet. (Учтите, что драйверы мини-порта WAN работают с подключениями по коммутируемой линии связи.)

```
lkd> .load ndiskd
Loaded ndiskd extension DLL
lkd> !miniports
NDIS Driver verifier level: 0
NDIS Failed allocations   : 0
Miniport Driver Block: 86880d78, Version 0.0
  Miniport: 868cf0e8, NetLuidIndex: 1, IfIndex: 9, RAS Async Adapter
Miniport Driver Block: 84c3be60, Version 4.0
   Miniport: 84c3c0e8, NetLuidIndex: 3, IfIndex: 15, VMware Virtual Ethernet
Adapter
Miniport Driver Block: 84c29240, Version 0.0
  Miniport: 84c2b438, NetLuidIndex: 0, IfIndex: 2, WAN Miniport (SSTP)
...
lkd> !miniport 84bcc0e8
 Miniport 84bcc0e8 : Broadcom NetXtreme 57xx Gigabit Controller, v6.0
    AdapterContext : 85f6b000
    Flags          : 0c452218
                     BUS_MASTER, 64BIT_DMA, IGNORE_TOKEN_RING_ERRORS
                     DESERIALIZED, RESOURCES_AVAILABLE, SUPPORTS_MEDIA_SENSE
                     DOES_NOT_DO_LOOPBACK, SG_DMA,
                     NOT_MEDIA_CONNECTED,
    PnPFlags       : 00610021
                     PM_SUPPORTED, DEVICE_POWER_ENABLED, RECEIVED_START
                     HARDWARE_DEVICE, NDIS_WDM_DRIVER,
    MiniportState        : STATE_RUNNING
    IfIndex                 : 10
    Ndis5MiniportInNdis6Mode : 0
    InternalResetCount    : 0000
    MiniportResetCount    : 0000
    References            : 5
```

```
UserModeOpenReferences: 0
PnPDeviceState        : PNP_DEVICE_STARTED
CurrentDevicePowerState : PowerDeviceD0
Bus PM capabilities
DeviceD1:        0
DeviceD2:        0
WakeFromD0:        0
WakeFromD1:        0
WakeFromD2:        0
WakeFromD3:        1
SystemState        DeviceState
PowerSystemUnspecified    PowerDeviceUnspecified
S0            D0
S1            PowerDeviceUnspecified
S2            PowerDeviceUnspecified
S3            D3
S4            D3
S5            D3
SystemWake: S5
    DeviceWake: D3
WakeupMethods Enabled 2:
    WAKE_UP_PATTERN_MATCH
WakeUpCapabilities:
MinMagicPacketWakeUp: 4
MinPatternWakeUp: 4
MinLinkChangeWakeUp: 0
Current PnP and PM Settings:          : 00000030
                DISABLE_WAKE_UP, DISABLE_WAKE_ON_RECONNECT,
Translated Allocated Resources:
    Memory: ecef0000, Length: 10000
    Interrupt Level: 9, Vector: a8
MediaType      : 802.3
DeviceObject   : 84bcc030, PhysDO : 848fd6b0  Next DO: 848fc7b0
MapRegisters   : 00000000
FirstPendingPkt: 00000000
DriverVerifyFlags  : 00000000
Miniport Interrupt : 85f72000
Miniport version 6.0
Miniport Filter List:
Miniport Open Block Queue:
  8669bad0: Protocol 86699530 = NDISUIO, ProtocolBindingContext 8669be88, v6.0
  86690008: Protocol 86691008 = VMNETBRIDGE, ProtocolBindingContext 866919b8, v5.0
  84f81c50: Protocol 849fb918 = TCPIP6, ProtocolBindingContext 84f7b930, v6.1
  84f7b230: Protocol 849f43c8 = TCPIP, ProtocolBindingContext 84f7b5e8, v6.1
```

Поле Flags изучаемого мини-порта показывает, что мини-порт поддерживает 64-разрядную операцию прямого доступа к памяти (64BIT_DMA), что сетевая среда в данный момент не активна (NOT_MEDIA_CONNECTED) и что он может

динамически обнаружить, когда сетевая среда подключена или когда она отключена (SUPPORTS_MEDIA_SENSE). Также перечисленные данные являются отображением состояния электропитания устройства от системы на ресурсы шины, которое диспетчер устройств Plug and Play назначил адаптеру. ■

Разновидности NDIS-драйверов мини-порта

Модель NDIS также поддерживает гибридные NDIS-драйверы сетевого транспорта, называемые промежуточными NDIS-драйверами. Эти драйверы находятся между транспортными драйверами и NDIS-драйверами мини-порта. С позиции NDIS-драйверов мини-порта промежуточный NDIS-драйвер похож на транспортный драйвер, а с позиции транспортного драйвера промежуточный NDIS-драйвер похож на NDIS-драйвер мини-порта. Промежуточные NDIS-драйверы могут видеть весь сетевой трафик, происходящий в системе, потому что эти драйверы находятся между драйверами протокола и сетевыми драйверами. На промежуточных NDIS-драйверах основано программное обеспечение, которое предоставляет сетевым адаптерам отказоустойчивость и сбалансированность нагрузки, например разработанный Microsoft выравниватель сетевой нагрузки — Network Load Balancing Provider. И наконец, в NDIS-модели также реализованы драйверы легких фильтров — lightweight filter drivers (LWF), — которые похожи на промежуточные драйверы, но разработаны специально для фильтрации сетевого трафика. LWF-драйверы поддерживают динамические вставки и удаления при работе протокольного стека. Фильтрующие драйверы могут фильтровать все входящие и выходящие коммуникации использующегося адаптера мини-порта. Они также могут выбирать определенные типы фильтрации (пакетов данных или управляющих сообщений) и могут пропускать мимо то, что их не интересует.

NDIS-драйверы, ориентированные на установку соединения

Поддержка сетевого оборудования, ориентированного на установку соединения (например, PPP), является характерной особенностью Windows, которая делает управление соединениями и их установку стандартом сетевой архитектуры Windows. NDIS-драйверы, ориентированные на установку соединения, используют множество таких же API-функций, которые используются стандартными NDIS-драйверами, но NDIS-драйверы, ориентированные на установку соединения, отправляют пакеты через установленные сетевые соединения, а не помещают их в сетевую среду.

Кроме мини-портовой поддержки, ориентированной на установку соединения сетевой среды, в NDIS включены определения для драйверов, работающих на поддержку драйвера мини-порта, ориентированного на установку соединения:

- Диспетчеры вызовов (call managers) являются NDIS-драйверами, которые предоставляют для клиентов, ориентированных на установку соединения, службы установки соединения и освобождения канала. Диспетчер вызовов использует мини-порт, ориентированный на установку соединения, для обмена управляющими (сигнальными) сообщениями с сетевыми коммутаторами или другой сетевой средой, ориентированной на установку соединения. Диспетчер

вызовов поддерживает один или несколько протоколов обмена сигналами и реализован в виде драйвера сетевого протокола.

- Интегрированный диспетчер вызовов мини-порта — Integrated miniport call manager (MCM) — является драйвером мини-порта, ориентированным на установку соединения, который также предоставляет службы диспетчера вызовов клиентам, ориентированным на установку соединения. По сути MCM является NDIS-драйвером мини-порта со встроенным диспетчером вызовов.
- Клиент, ориентированный на установку соединения использует службы установки соединения и освобождения канала диспетчера вызовов или MCM и службы отправки и получения NDIS-драйвера мини-порта, ориентированного на установку соединения. Клиент, ориентированный на установку соединения, может предоставить более высоким уровням сетевого стека свои собственные службы протокола, или же он может реализовать уровень эмуляции, служащий интерфейсом между старыми протоколами, не ориентированными на установку соединения, и сетевой средой, ориентированной на такую установку.

Взаимоотношения между этими компонентами показаны на рис. 7.39.

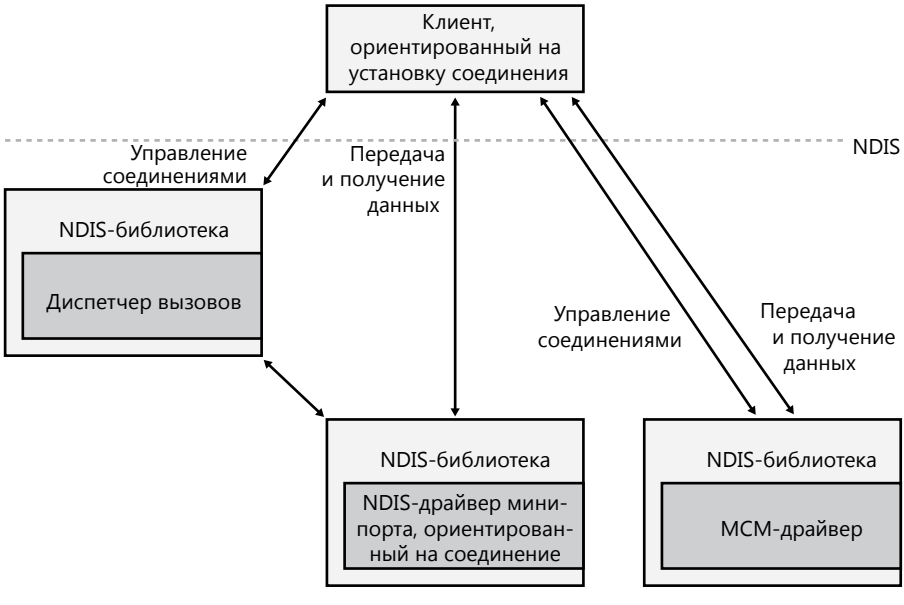


Рис. 7.39. NDIS-драйверы, ориентированные на установку соединения

ЭКСПЕРИМЕНТ: ИСПОЛЬЗОВАНИЕ NETWORK MONITOR ДЛЯ ЗАХВАТА СЕТЕВЫХ ПАКЕТОВ

Microsoft предоставляет инструментальное средство Network Monitor, позволяющее захватывать пакеты из одного или нескольких NDIS-драйверов мини-портов вашей системы путем установки NDIS-драйвера легкого фильтра (Netmon). Самую последнюю версию Network Monitor можно получить по адресу http://www.microsoft.com/download/en/details.aspx?id=4865. Не забудьте загрузить парсер протокола NetMon с адреса http://nmparsers.

codeplex.com/, в противном случае вы не сможете декодировать протоколы Microsofts. При первом запуске Network Monitor будет показано окно, похожее на то, что показано на рис. 7.40.

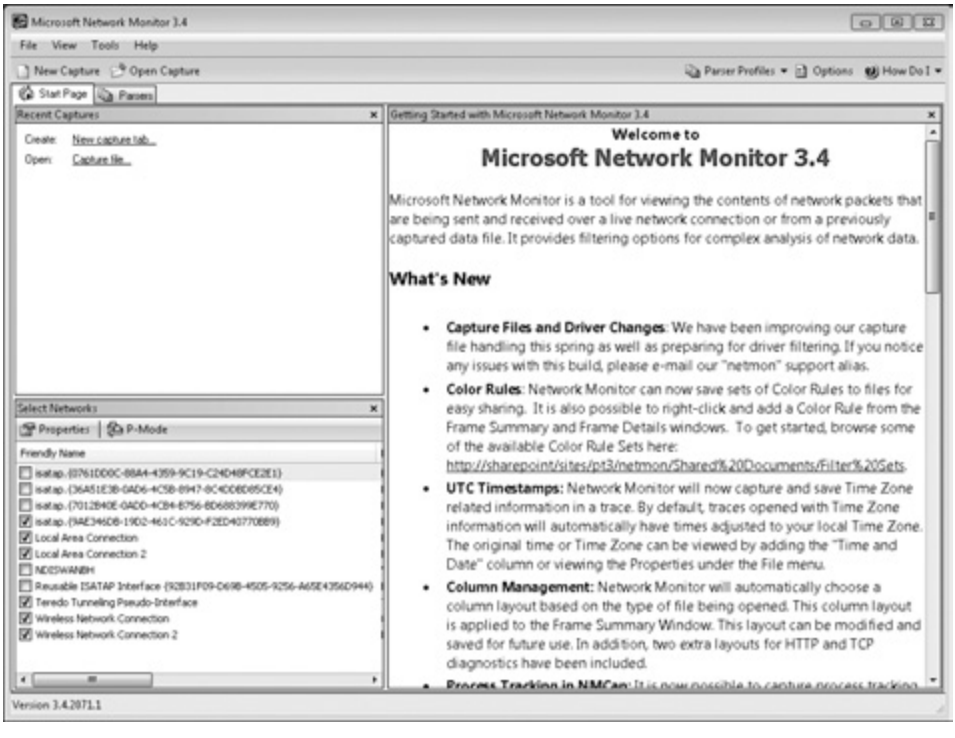


Рис. 7.40. Утилита Network Monitor

В панели Select Networks (Выбор сетей) Network Monitor позволяет выбрать то сетевое подключение, за которым нужно следить. После выбора одной или нескольких сетей щелчком на кнопке New Capture (Новый захват) на панели инструментов запустите среду захвата. Теперь можно инициировать отслеживание, щелкнув на кнопке Start (Старт) на панели инструментов. Выполните действия, приводящие к сетевой активности на отслеживаемом подключении (например, просмотр веб-страниц), и после того как вы увидите, что Network Monitor захватил пакеты, остановите отслеживание щелчком на кнопке Stop (Стоп). В области окна Frame Summary (Сводка сведений о кадре) вы увидите весь необработанный сетевой трафик за период действия захвата. В области окна Network Conversations (Сетевой диалог) выводится сетевой трафик, изолированный, где это можно, процессом. В данном примере после щелчка на процессе Iexplore.exe Network Monitor показывает в области окна Frame Summary (Сводка сведений о кадре) только те кадры, которые относятся к этому процессу (рис. 7.41).

В окне показаны HTTP-пакеты, которые Network Monitor захватил при обращении к веб-сайту компании Microsoft из браузера Internet Explorer. Если щелкнуть на кадре, Network Monitor показывает вид пакета, который разбит на части, чтобы показать в области окна Frame Details (Подробности

кадра), которую можно увидеть на предыдущей копии экрана, различные многоуровневые заголовки приложений и протоколов.

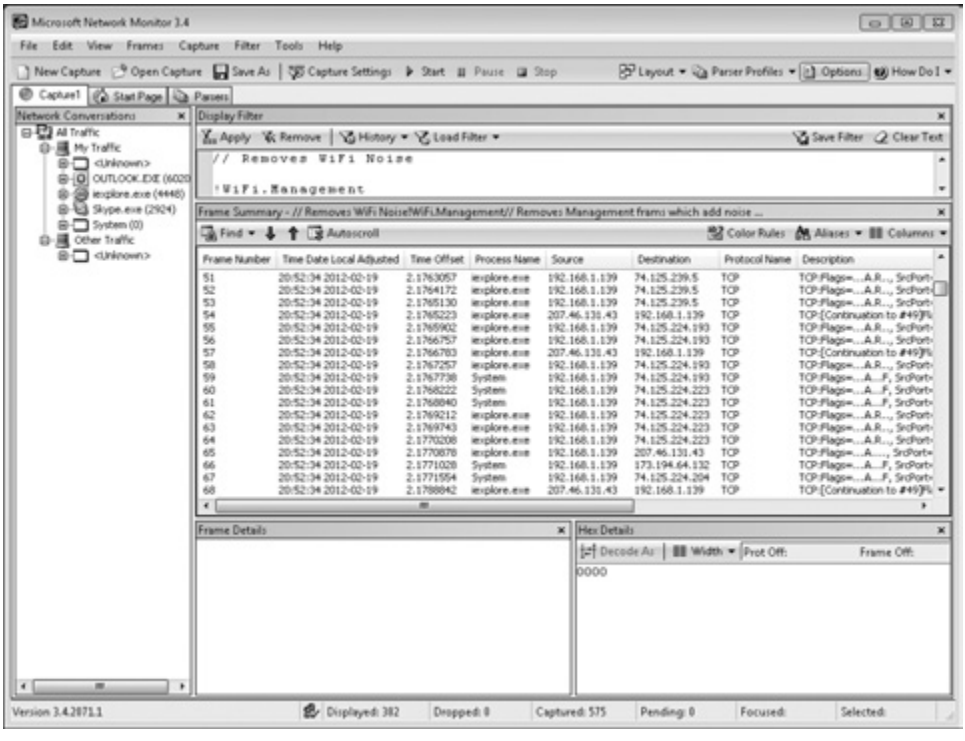


Рис. 7.41. Захват пакетов с помощью Network Monitor

Network Monitor имеет также множество других свойств, например инициаторы и фильтры захвата, которые делают его эффективным средством для выявления причин возникающих сетевых проблем. Можно также добавлять парсеры для других протоколов, просматривать исходный код этих протоколов и вносить в него изменения. Парсеры Network Monitor находятся на сайте принадлежащего Microsoft проекта с открытыv кодом CodePlex (http://nmparsers.codeplex.com). ■

Remote NDIS

До разработки Remote NDIS производитель, разрабатывающий сетевое USB-устройство, должен был предоставлять драйвер, который служил интерфейсом с NDIS в качестве драйвера мини-порта, а также служил интерфейсом с драйвером шины USB WDM (рис. 7.42).

Remote NDIS является спецификацией для сетевых USB-устройств. Эта спецификация устраняет необходимость написания производителем оборудования NDIS-драйвера мини-порта, определяя сообщения и механизм, с помощью которого сообщения передаются по USB. Сообщения Remote NDIS зеркально отображают NDIS-интерфейс и включают сообщения для инициализации и перезапуска устройства, передачи и приема пакетов, установки и запроса параметров устройства и индикации состояния канала связи.

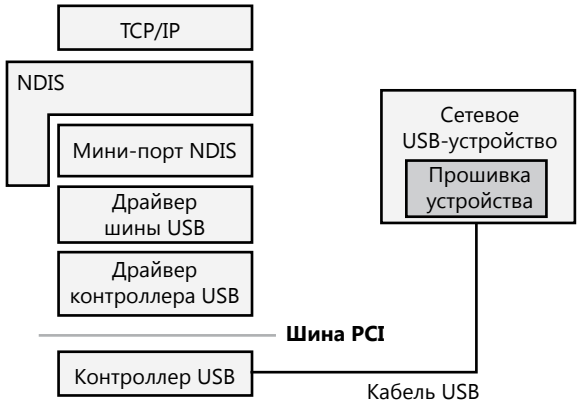


Рис. 7.42. NDIS-драйвер мини-порта для сетевого USB-устройства

Архитектура Remote NDIS, показанная на рис. 7.43, зависит от поставляемого Microsoft NDIS-драйвера мини-порта, %SystemRoot%\System32\Drivers\Rndismp.sys, который преобразует NDIS-команды и переправляет их USB-устройству. Архитектура позволяет одному NDIS-драйверу мини-порта использоваться для всех Remote NDIS USB-устройств.

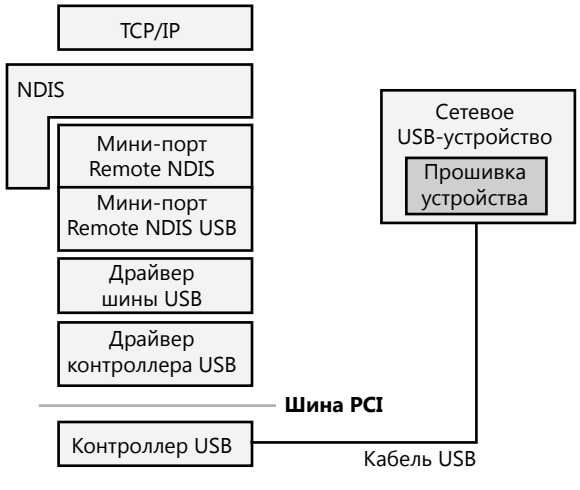


Рис. 7.43. Архитектура Remote NDIS для сетевых USB-устройств

В настоящее время USB является единственной шиной, поддерживаемой RNDIS в Windows.

QoS

Если не предпринять какие-либо специальные меры, сетевой IP-трафик доставляется по принципу «первым пришел, первым и обслужен». Приложения не имеют средств управления приоритетностью своих сообщений и могут испытывать на себе пульсирующее поведение сети, при котором время от времени они полу-

чают высокую пропускную способность и низкий уровень задержек, чередующиеся с получением низкого уровня сетевой производительности. Конечно, такой уровень обслуживания в большинстве ситуаций вполне допустим (например, при передаче файлов или просмотре интернета), но число сетевых приложений, требующих более постоянных уровней обслуживания или гарантий качества обслуживания — Quality of Service (QoS), — постоянно растет. Примерами приложений, требующих постоянного уровня сетевой производительности, могут послужить видеоконференции, разноформатные потоки данных и программы планирования ресурсов предприятий — enterprise resource planning (ERP). QoS позволяет приложениям определить минимальную пропускную способность и максимальные показатели задержек, которые могут быть удовлетворены только в том случае, если каждое сетевое программное обеспечение и аппаратный компонент между отправителем и получателем поддерживают стандарты QoS[1].

Windows поддерживает стандарт QoS через его реализацию на основе политики, которая всецело использует сетевой стек TCP/IP нового поколения (Next Generation TCP/IP network stack), WFP и NDIS-драйверы легких фильтров. Реализация допускает управление или определение приоритетности использования пропускной способности на основе различных условий, например приложения, IP-адреса источника или приемника, используемого протокола и портов источника или приемника. Сетевые администраторы обычно применяют настройки QoS к сеансу входа в систему или к компьютеру с групповой политикой на основе Active Directory, но они также могут применяться локально.

QoS на основе политики предоставляет два метода, с помощью которых можно управлять пропускной способностью. Первый метод использует специальное поле в IP-заголовке, которое называется точкой кода дифференцированных услуг — Differentiated Services Code Point (DSCP). Маршрутизаторы, поддерживающие DSCP, считывают значение и разделяют пакеты на очереди с определенными приоритетами. Архитектура QoS в Windows может помечать исходящие пакеты соответствующим DSCP-полем, чтобы сетевые устройства могли предоставить различные уровни обслуживания. Другой метод управления пропускной способностью заключается в возможности простого дросселирования исходящего трафика на основе ранее обозначенных условий, где компоненты QoS ограничивают пропускную способность до указанной скорости.

Как показано на рис. 7.44, имеющаяся в Windows реализация QoS состоит из нескольких компонентов. Первый из них, расширение QoS на стороне клиента (%SystemRoot%\System32\Gptext.dll), уведомляет клиента групповой политики и инспекционный модуль (Inspection Module) QoS о том, что настройки QoS изменились. Следующий, инспекционный модуль QoS (Качество обслуживания предприятия, Enterprise Quality of Service — eQoS), являющийся WFP-компонентом инспектирования пакетов, реализованным в TCP/IP-драйвере, реагирующим на изменения политики, извлекает обновленную политику и работает транспортным уровнем и планировщиком пакетов — QoS Packet Scheduler, — для того чтобы пометить трафик, соответствующий политике. И наконец, планировщик пакетов, QoS Packet Scheduler, или Pacer (%SystemRoot%\System32\

[1] Например, IEEE 802.1P, промышленный стандарт, определяющий формат QoS-пакетов и реакцию на них устройств сетевой модели OSI второго уровня (коммутаторов и сетевых адаптеров).

Drivers\Pacer.sys), предоставляющий такие функции легкого NDIS-фильтра, как дросселирование и установку DSCP-значения для управления планированием пакетов на основе QoS-политик. Pacer также предоставляет поддержку GQoS (Generic QoS — общих для QoS) и TC (Traffic Control — управляющих трафиком) API-функций для устаревших Windows-приложений, использующих эти механизмы.

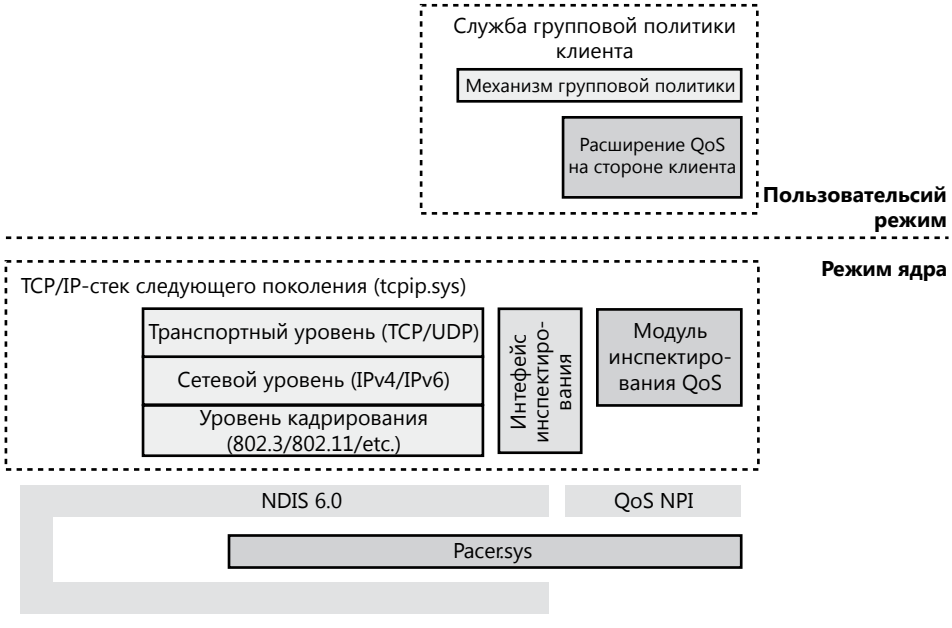


Рис. 7.44. Архитектура механизма QoS, основанная на политике

Кроме основанной на политике общесистемной поддержки QoS, предоставляемой архитектурой QoS, Windows допускает у определенных классов приложений, основанных на сокетах, наличие индивидуального специфического управления поведением QoS через API-интерфейс, называемый механизмом подстройки аудио-видео качества Windows — Quality Windows Audio/Video Experience, или qWAVE.

Мультимедиа-приложения, основанные на применении сети, например Voice over IP (VoIP), могут использовать qWAVE API для запроса информации о пропускной способности сети в реальном масштабе времени и для адаптации к изменениям сетевых условий, а также для назначения приоритетов пакетам для эффективного использования доступной пропускной способности. Механизм qWAVE также использует ранее рассмотренные топологические протоколы для динамического определения, поддерживают ли текущие сетевые устройства требуемую пропускную способность, к примеру, для видеопотока. Этот механизм может уведомить приложения о снижении пропускной способности, обозначив, к примеру, момент ожидаемого снижения мультимедийным приложением качества потока.

Механизм qWAVE реализован в API-библиотеке QoS2 (%SystemRoot%\System32\Qwave.dll) и предоставляет четыре основных компонента:

- Контроль приема (admission control), определяющий момент запуска нового сетевого мультимедийного потока, при условии, что текущая сеть способна поддерживать запрошенную устойчивую полосу пропускания.
- Кэширование, позволяющее пропускать детальные проверки контроля приема, если в прошлом уже были подобные схемы использования и вычисленные результаты уже были кэшированы.
- Мониторинг и зондирование, отслеживающие доступную пропускную способность и уведомляющие приложения в ходе ее снижения или повышения времени отклика.
- Тегирование и формирование трафика, где используются ранее упомянутые технологии 802.11p и DSCP для тегирования пакетов с соответствующим уровнем приоритета, чтобы обеспечить своевременную доставку.

На рис. 7.45 показан общий вид архитектуры qWAVE.

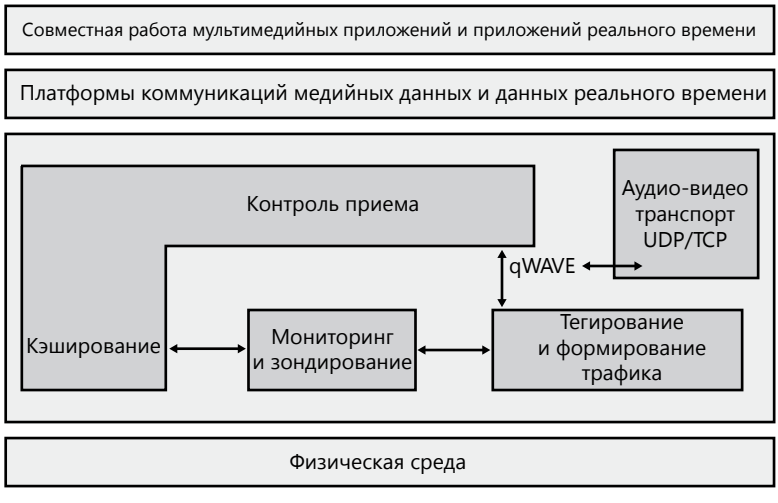


Рис. 7.45. Архитектура qWAVE

Привязка

Завершающей частью пазла сетевой архитектуры Windows является способ, с помощью которого компоненты на разных уровнях — уровне сетевого API, уровне транспортного драйвера, уровне NDIS-драйвера — находят друг друга. Процесс, связывающий уровни, называется *привязкой*. Наличие привязки можно было бы проследить при изменении сетевой конфигурации путем добавления или удаления компонентов, используя папку сетевых подключений.

При установке сетевого компонента его нужно снабдить INF-файлом. Этот файл включает каталоги, которых должны придерживать процедуры установочного API при установке и настройке компонента, включая привязку зависимостей или привязку отношений. Разработчик может указать зависимости привязки для собственного компонента, чтобы диспетчер управления службами — Service Control Manager (см. главу 4) — не просто загрузил компонент в правильном

порядке, но загрузил его только при том условии, что в системе присутствуют другие зависимые компоненты. Привязка отношений, которые механизм привязки определяет с помощью дополнительной информации в INF-файле компонента, устанавливает связи между компонентами на различных уровнях. Эти связи определяют, какие компоненты сетевой компонент одного уровня может использовать из того уровня, который находится под ним.

Например, служба Рабочая станция (редиректор) автоматически привязывается к протоколу TCP/IP. Порядок привязки, который можно исследовать во вкладке Адаптеры и привязки (Adapters And Bindings) диалогового окна Дополнительные параметры (Advanced Settings) (рис. 7.46), определяет приоритет привязки[1]. Когда редиректор получает запрос на доступ к удаленному файлу, он отправляет запрос одновременно обоим драйверам протокола. Когда поступает ответ, редиректор ждет, пока он тоже не получит ответы от любого протокола более высокого уровня. Только потом редиректор вернет результат вызывающей процедуре. Следовательно, изменение порядка привязки может принести пользу, поскольку привязки с более высоким приоритетом могут также оказаться более эффективными с точки зрения производительности или более подходящими большинству компьютеров вашей сети. Используя диалоговое окно Дополнительные параметры (Advanced Settings), можно также самостоятельно удалять привязки.

Рис. 7.46. Редактирование привязок с помощью диалогового окна Дополнительные параметры (Advanced Settings)

Информация о привязке компонента хранится в параметре Bind раздела Linkage, который находится в разделе реестра, относящемся к сетевому компоненту. Например, если исследовать раздел HKLM\SYSTEM\CurrentControlSet\

[1] Порядок запуска диалогового окна Дополнительные параметры был показан в разделе «Поддержка нескольких редиректоров».

Services\LanmanWorkstation\Linkage\Bind, можно увидеть информацию привязки для службы Рабочая станция (Workstation).

Многоуровневые сетевые службы

В Windows есть сетевые службы, которые построены на API-интерфейсах и компонентах, представленных в этой главе. Описание возможностей и подробностей внутренней реализации всех этих служб выходит за рамки данной книги, но в этом разделе предоставляется краткий обзор удаленного доступа, Active Directory, балансировки сетевой нагрузки и распределенной файловой системы — Distributed File System (DFS), — включая репликацию DFS — DFS Replication (DFSR).

Удаленный доступ

Удаленный доступ, получить который можно с помощью Windows Server со службой Маршрутизация и удаленный доступ (Routing and Remote Access), позволяет клиентам удаленного доступа подключиться к серверам удаленного доступа и получить доступ к таким сетевым ресурсам, как файлы, принтеры и сетевые службы, точно так же, как если бы клиент был физически подключен к сети сервера удаленного доступа. Windows предоставляет два вида удаленного доступа:

- Коммутируемый удаленный доступ, используемый клиентами, подключающимися к серверу удаленного доступа с помощью телефона или другой телекоммуникационной инфраструктуры. Телекоммуникационная среда используется для создания временного физического или виртуального соединения между клиентом и сервером.
- Удаленный доступ по виртуальной частной сети — virtual private network (VPN), — позволяющий VPN-клиенту установить виртуальное двухточечное подключение к серверу через IP-сеть, например через Интернет. Windows также поддерживает протокол передачи с использованием безопасных сокетов — Secure Socket Transmission Protocol (SSTP), — являющийся новым туннелированным протоколом для VPN-соединений, имеющих возможность проходить через большинство брандмауэров и маршрутизаторов, которые блокируют PPTP- или L2TP/IPsec-трафик. Это делается за счет упаковки PPP-данных по SSL-каналу HTTPS-протокола. Поскольку последний работает с портом 443 и является частью обычного поведения при просмотре Интернета, его доступность более вероятна, чем традиционные VPN-протоколы туннелирования.

Удаленный доступ отличается от решений по дистанционному управлению, поскольку удаленный доступ работает как прокси-подключение к сети Windows, а программное обеспечение дистанционного управления выполняет приложения на сервере, предоставляя клиенту пользовательский интерфейс.

Active Directory

Active Directory является Windows-реализацией служб каталогов (RFC 4510) облегченного протокола доступа к каталогам — Lightweight Directory Access

Protocol (LDAP). По своей сути Active Directory является базой данных, в которой хранятся объекты, представляющие ресурсы, определенные приложениями в сети Windows. Например, в Active Directory хранится структура домена Windows и сведения о его участниках, включая информацию о пользовательских учетных записях и паролях.

Классы объектов и атрибуты, определяющие свойства объектов, указываются схемой. Объекты в Active Directory имеют иерархическую организацию, очень похожую на логическую организацию реестра, где в объектах-контейнерах могут храниться другие объекты, включая другие объекты-контейнеры (см. главу 6).

Active Directory поддерживает множество API-интерфейсов, которые могут использоваться клиентами для доступа к объектам в базе данных Active Directory:

- LDAP C API, являющийся API на языке C, использующим сетевой протокол LDAP. Приложения, написанные на C или C++ могут использовать этот API напрямую, а приложения, написанные на других языках, могут получить к нему доступ через уровни трансляции.
- Служебные интерфейсы Active Directory — Active Directory Service Interfaces (ADSI), — являющиеся COM-интерфейсом к Active Directory, реализованным поверх LDAP, который абстрагирует от подробностей LDAP-программирования. ADSI поддерживает несколько языков, включая Microsoft Visual Basic, C и Microsoft Visual C++. ADSI может также использоваться приложениями Microsoft Windows Script Host (WSH).
- API сообщений — Messaging API (MAPI), — который поддерживается в целях совместимости к приложениями-клиентами Microsoft Exchange и Outlook Address Book.
- API-интерфейсы администратора учетных данных в системе защиты — Security Account Manager (SAM), — которые являются надстройкой над Active Directory, предоставляющей интерфейс для пакетов аутентификации при входе в систему, например для MSV1_0 (%SystemRoot%\System32\Msv1_0.dll, использующемуся для устаревшего диспетчера аутентификации NT LAN) и Kerberos (%SystemRoot%\System32\Kdcsvc.dll).
- Сетевые API-интерфейсы Windows NT (Net API), которые используются клиентами Windows NT 4 для получения доступа к Active Directory через SAM.
- NTDS API, который используется для поиска SID-идентификаторов и GUID-идентификаторов в реализации Active Directory (в основном через DsCrackNames), а также по своему основному предназначению, для управления Active Directory и его тиражирования. Рядом сторонних разработчиков были созданы приложения, проводящие мониторинг Active Directory с помощью этих API-интерфейсов.

Продукт Active Directory реализован в виде файла базы данных, исходное имя которого %SystemRoot%\Ntds\Ntds.dit, и тиражируется по доменным контроллерам домена. Служба каталогов Active Directory, которая является службой Windows, выполняемой в процессе подсистемы проверки подлинности локальной системы безопасности — Local Security Authority Subsystem (LSASS) — и управляющей базой данных, используя DLL-библиотеки, реализующие дисковую структуру базы данных, а также предоставляющие обновления на основе тран-

закций для защиты целостности базы данных. Хранилище базы данных Active Directory основано на версии расширяемого механизма хранения — Extensible Storage Engine (ESE), — также известного, как JET Blue, базы данных, используемой Microsoft Exchange Server 2007, поисковой системой рабочего стола — Desktop Search — и почтой Windows Mail. Библиотека ESE (%SystemRoot%\System32\Esent.dll) предоставляет процедуры для доступа к базе данных, открытые для других приложений, которые также могут их использовать. Архитектура Active Directory показана на рис. 7.47.

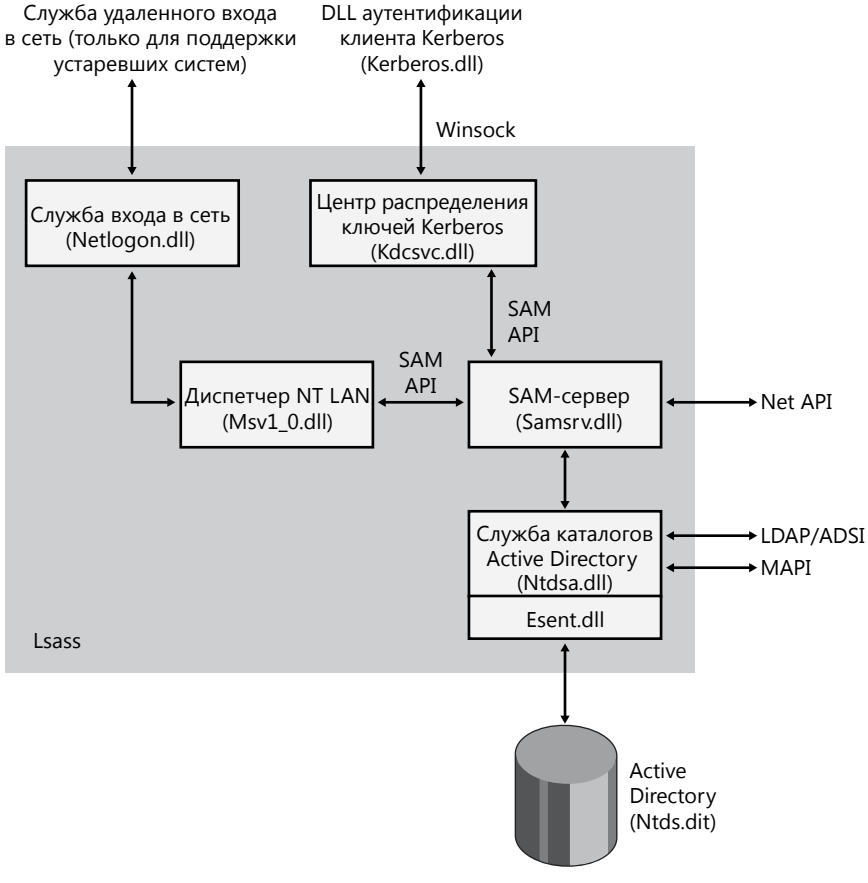


Рис. 7.47. Архитектура Active Directory

Network Load Balancing

Как уже говорилось ранее в этой главе, балансировка сетевой нагрузки — Network Load Balancing, — которая включается в серверные версии Windows, основана на технологии легких фильтров NDIS. Network Load Balancing допускает создание кластера, содержащего до 32 компьютеров, которые называются *узлами кластера*. Кластер может поддерживать несколько выделенных IP-адресов и один виртуальный IP-адрес, опубликованный для доступа клиентов. Клиентские запросы

поступают всем компьютерам кластера, но отвечает на запрос только один узел кластера. NDIS-драйверы балансировки сетевой нагрузки эффективно делят клиентское пространство между доступными узлами кластера распределительным образом. Таким образом, каждый узел обрабатывает свою часть входящих клиентских запросов, и каждый клиентский запрос всегда обрабатывается одним и только одним узлом. Узел кластера, определивший, что ему нужно обработать клиентский запрос, позволяет запросу распространяться вверх, к драйверу протокола TCP/IP, и, в конечном счете, к серверному приложению, но не к другому узлу кластера. Если узел кластера дает сбой, оставшиеся узлы кластера понимают, что этот узел кластера больше не служит кандидатом на обработку запросов, и перераспределяют входящие клиентские запросы на оставшиеся узлы кластера. Больше клиентские запросы отказавшему узлу кластера не отправляются. В качестве замены к кластеру может быть добавлен другой узел кластера, и затем он без каких-либо коллизий начнет обрабатывать клиентские запросы.

Network Load Balancing не является универсальным кластерным решением, потому что серверное приложение, с которым связываются клиенты, должно обладать рядом характеристик: во-первых, оно должно основываться на протоколах, поддерживаемых TCP/IP-стеком Windows, и во-вторых, оно должно уметь обрабатывать клиентские запросы на любой системе, участвующей в Network Load Balancing. Это второе требование обычно означает, что приложение, которое должно иметь доступ к общему состоянию с целью обслуживания клиентских запросов, должно само управлять общим состоянием, поскольку Network Load Balancing не включает в себя службы для автоматического распределения общего состояния между узлами кластера. Приложения, идеально подходящие для Network Load Balancing, — веб-сервер, который обслуживает статическое содержимое, Windows Media Server, и службы терминалов. Пример работы механизма балансировки сетевой показан на рис. 7.48.

Защита сетевого доступа

Одним из самых больших испытаний, с которыми сталкиваются сетевые администраторы, — это обеспечение того, что системы, подключающиеся к их частным сетям, соответствуют требованиям времени и отвечают требованиям политики состояния работоспособности, проводимой в их организации. Политика состояния работоспособности включает специфические требования, которым должна отвечать система, например минимально требуемые обновления системы или минимально требуемую версию вирусных сигнатур. Навязывать эти требования еще сложнее, когда системы, например домашние компьютеры или ноутбуки, не находятся под контролем сетевого администратора. Атакующие часто создают вредоносные программы, нацеленные на устаревшее программное обеспечение, поэтому пользователи, не поддерживающие свои системы на уровне современных требований, не устанавливающие самые последние обновления операционной системы или вирусные сигнатуры, подвергают частные сети организаций риску подверженности атакам или проникновения вирусов.

Защита сетевого доступа — Network Access Protection (NAP) — предоставляет механизм, помогающий сетевым администраторам навязать всем системам, тре-

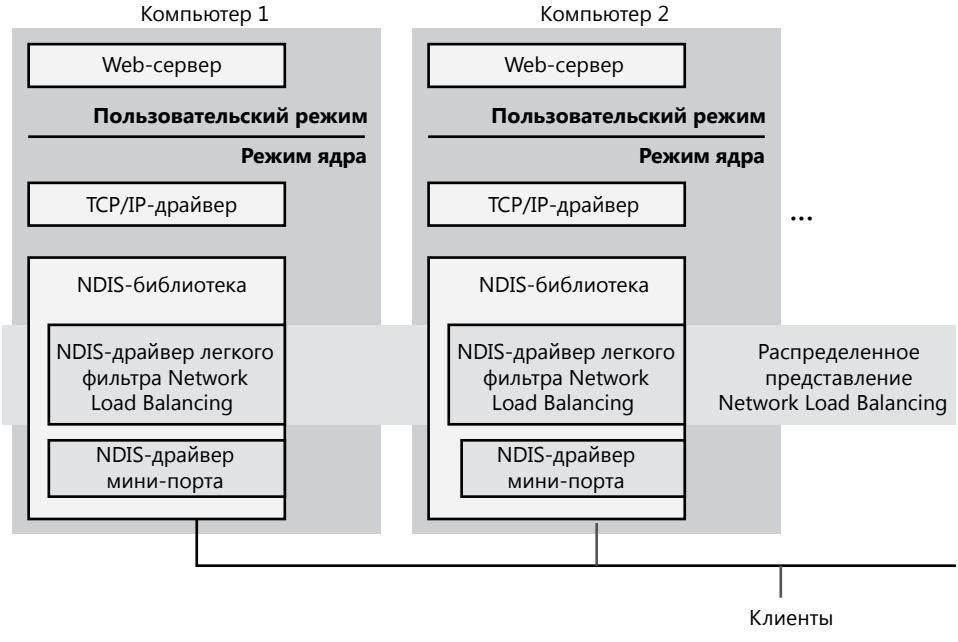


Рис. 7.48. Работа механизма балансировки сетевой нагрузки

бующим доступ к сети, соответствие политикам состояния работоспособности. Системы, не отвечающие требуемым политикам состояния работоспособности, изолируются от сети и помещаются в карантин. Когда несовместимые системы находятся в карантине, на их сетевое подключение накладываются строгие ограничения, и они могут только видеть корректирующие серверы, от которых можно получить необходимые обновления для придания им совместимости. Тем самым гарантируется, что доступ к сети организации разрешен только тем системам, которые отвечают требованиям политик состояния работоспособности. Механизм NAP не рассчитан на защиту сети от злоумышленников, он рассчитан на помощь администраторам в обеспечении нужного состояния работоспособности систем, находящихся в сети, что, в свою очередь, помогает поддерживать общую целостность сети. NAP является неоднородной системой, имеющей клиентов, работающих на других операционных системах, например Mac OS X и Linux, и имеющей ряд сторонних агентов состояния работоспособности системы (System Health Agents), механизмов проверки состояния работоспособности (System Health Validators) и клиентов принуждения.

Полное описание NAP выходит за рамки этой книги, но на рис. 7.49 и 7.50 показаны различные компоненты, реализующие NAP на клиентских и серверных системах. Подробное описание NAP можно найти по адресу http://technet.microsoft.com/en-us/network/bb545879.aspx.

Вкратце, NAP на стороне клиента состоит из следующих компонентов:

- **Агент состояния работоспособности системы — System Health Agent (SHA).** Следит за одним или несколькими аспектами состояния работоспособности клиента и предоставляет одно или несколько заявлений о состоянии рабо-

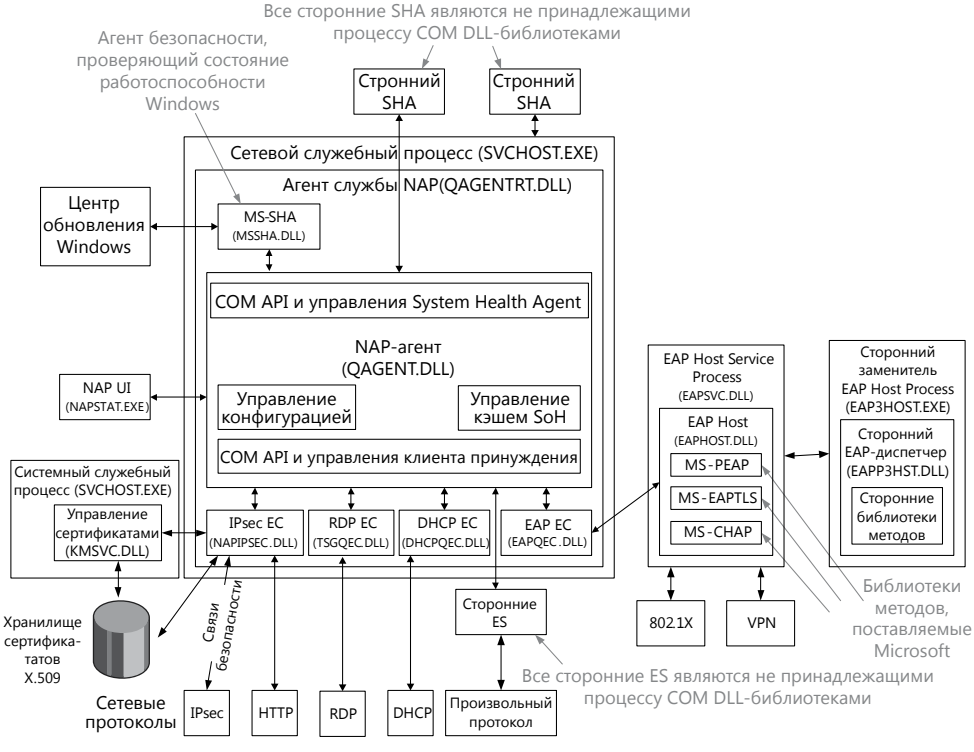


Рис. 7.49. Архитектура NAP на стороне клиента

тоспособности — Statements of Health (SoH) — для NAP-агента локальной системы. Например, антивирусный SHA может проверять номера версий антивирусных механизмов и файлов вирусных сигнатур и помещать эту информацию в свое SoH. SHA может согласовываться с корректирующим сервером, чтобы несовместимые системы знали, как можно стать совместимыми. Например, SHA для проверки сигнатур антивируса может согласовываться с сервером, содержащим самый новый файл вирусных сигнатур, и с пакетом антивирусных приложений. Некоторые SHA-агенты не нуждаются в согласовании с корректирующим сервером. Например, SHA может просто сообщать настройкам локальной системы, что механизм проверки состояния работоспособности системы — System Health Validator (SHV), — запущенный на NAP-сервере SHV, можно использовать для определения того, включен или нет брандмауэр системы. В Windows XP с пакетом обновления 3 и в более новых версиях предоставляется SHA (%SystemRoot%\System32\Mssha.dll), который следит за настройками Центра поддержки — Windows Action Center (SHA-WAC). Этот SHA обычно называют Windows SHA, или WSH. Чтобы написать SHA, загляните в API-интерфейсы INapSystemHealthAgentBinding2, INapSystemHealthAgentCallback и INapSystemHealthAgentRequest. SHA зависит от System Health Validator (SHV), и ожидается, что автор SHA также предоставляет и SHV.

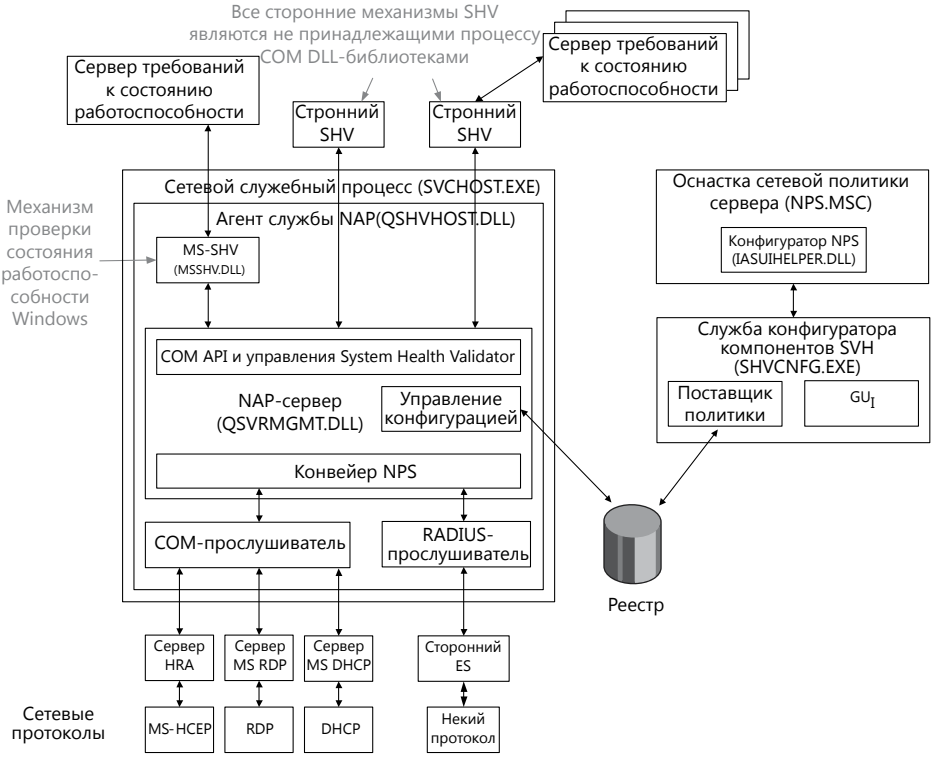


Рис. 7.50. Архитектура NAP на стороне клиента

ПРИМЕЧАНИЕ

Производители SHA должны понимать, что процесс оценки может произойти до получения системой IP-адреса (например, с помощью 802.1x), поэтому SHA не может искать данные за пределами клиентской системы. Кроме того, IP-адрес может измениться в любой момент времени (например, если NAP заставит клиента переместиться в карантинную VLAN), поэтому SHA не будет кэшировать сокеты или строить предположения относительно их IP-адреса.

- **Агент защиты сетевого доступа — NAP Agent.** %SystemRoot%\System32\qagentRT.dll (служба карантинного агента времени выполнения) запускается на каждом клиентском компьютере, собирает SoH от каждого SHA и передает эту информацию на NAP-сервер. NAP-агент связывается с NAP-сервером на сервере сетевой политики (Network Policy Server) с помощью протокола состояния работоспособности — Microsoft Statement of Health protocol [MS-SoH].
- **Клиент принуждения — Enforcement Client (EC).** Отвечает за связь с пунктом принуждения — Enforcement Point — при попытке подключения к сети и за принуждение машины к соответствию NAP-политикам. Пункт принуждения является сервером или устройством сетевого доступа, которое может использоваться с NAP для запроса оценки состояния работоспособности клиента в соответствии с требованиями NAP и предоставить ограниченный сетевой

доступ или обмен данными. Если состояние работоспособности машины не соответствует требованиям, NAP EC обозначает NAP-агенту ограниченный статус. Windows предоставляет EC-клиентов для IPsec (%SystemRoot%\System32\NapIPsec.dll), 802.1X и VPN аутентификационным подключениям EAP (%SystemRoot%\System32\Eapqec.dll), DHCP (%SystemRoot%\System32\Dhcpqec.dll) и шлюзу удаленного рабочего стола — Remote Desktop gateway (%SystemRoot%\System32\Tsgqec.dll). Чтобы написать EC, посмотрите на API-интерфейсы INapEnforcementClientBinding, INapEnforcementClientCallback и INapEnforcementClientConnection2.

ПРИМЕЧАНИЕ

Название «клиент принуждения» может показаться непонятным. Название ссылается на его роль в качестве клиента сетевого пункта принуждения, поэтому оно больше относится к тому, как клиентская система получает доступ к сети (хотя управление доступом является, как правило, лишь частью функции этого клиента).

На следующей схеме показаны серверные NAP-компоненты. На серверной стороне весь механизм является дополнением к серверу сетевой политики — Network Policy Server (NPS), — который является частью службы IAS. Вообще-то, запросы на состояние работоспособности поступают в NPS в качестве дополнения к RADIUS-запросам, отправленным NPS пунктом принуждения. Затем NPS передает заявление о состоянии работоспособности — Statement of Health (SoH) — серверам на уровень проверки этого состояния, а они передают SoH соответствующему SHV.

С точки зрения NPS, запросы поступают от RADIUS-клиентов (например, от сетевого коммутатора 802.1x, VPN-сервера, DHCP-сервера и т. д.) в пакетах RADIUS UDP. Или разрешаются частные ALPC-вызовы[1]. Спецификация RADIUS (RFC 2865) предоставляется для максимального размера пакета в 4096 байт, что существенно влияет на объем данных, который SHA может отправить.

IPsec EC на стороне клиента общается с сервером авторизации работоспособности — Health Registration Authority (HRA) — через HTTP. HRA является IIS ISAPI фильтром, который передает SoH в адрес NPS (используя ALPC-интерфейс) и отвечает за выдачу сертификатов (когда машина квалифицируется для сертификата). Список серверов HRA может быть настроен с помощью DNS, путем добавления записи HRA-сервера и настройки клиента на получение списка от DNS. Сторонние производители могут реализовать RADIUS-клиента для общения с NPS через UDP.

- **Механизм проверки состояния работоспособности — System Health Validator (SHV).** Оценивает SoH-заявление, полученное от соответствующего SHA на стороне клиента, и определяет, совместим ли клиент с политикой состояния работоспособности организации, для чего производится проверка сервером требований к состоянию работоспособности — Health Requirements Server (HRS). Например, требования HRS для антивируса могут определять минимальный номер версии антивирусного механизма и версии файла сигнатур вирусов.

[1] Вместо прохождения через UDP, ALPC используется другими ролями Windows Server, например DHCP-сервером, для упрощения модели программирования.

ПРИМЕЧАНИЕ

Присутствие сервера требований к состоянию работоспособности — Health Requirements Server — является подробностью реализации; SHV может выполнить всю необходимую работу самостоятельно

SHV использует эту информацию для определения, соответствует ли SoH предоставленное SHA клиента, политике состояния работоспособности, предоставленной HRS. Чтобы написать SHV, следует обратить внимание на API-интерфейсы INapSystemHealthValidator и INapSystemHealthValidationRequest2. SHV зависит от агента состояния работоспособности системы — System Health Agent (SHA), — и ожидается, что автор SHA также предоставит и SHV.

На схеме не показан один или несколько корректирующих серверов (Remediation Server), позволяющих клиенту обрести совместимость. SHV не подключен к корректирующим серверам, но он знает об их существовании (что настраивается в административном порядке). Он передает информацию о серверах клиенту, когда SoH показывает, что этот клиент не совместим с текущими требованиями политики.

Настройка NAP-клиента проводится, как правило, в редакторе групповой политики с помощью оснастки Клиенты принудительной защиты (Enforcement Client), но ее можно также провести, используя MMC-оснастку Конфигурация клиента NAP (NAP client configuration) (%SystemRoot%\System32\Napclcfg.msc) или сетевую оболочку (%SystemRoot%\System32\Netsh.exe), как показано на рис. 7.51–7.53.

ПРИМЕЧАНИЕ

Групповая политика всегда имеет преимущество над другими настройками конфигурации, затем следует локальная конфигурация, а затем конфигурация автообнаружения DNS.

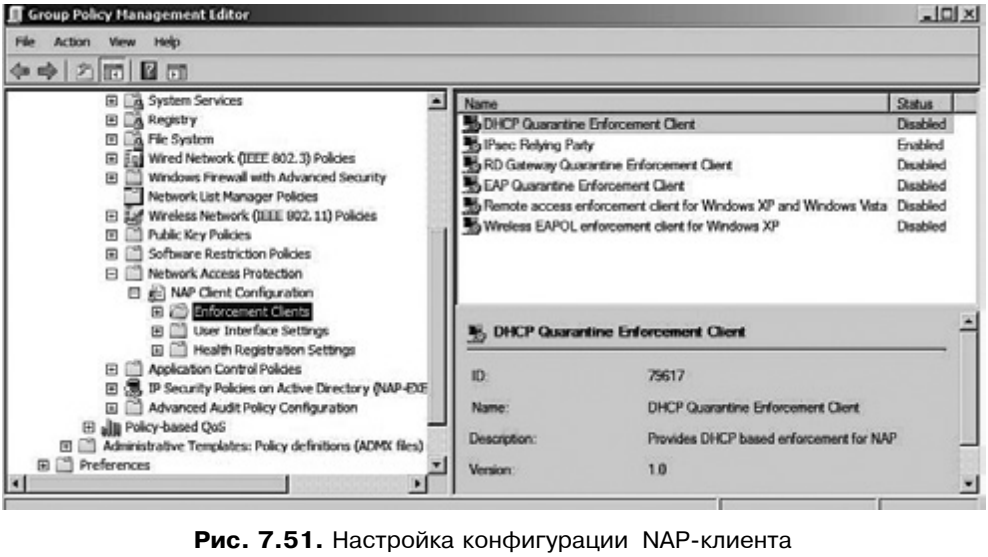


Рис. 7.51. Настройка конфигурации NAP-клиента

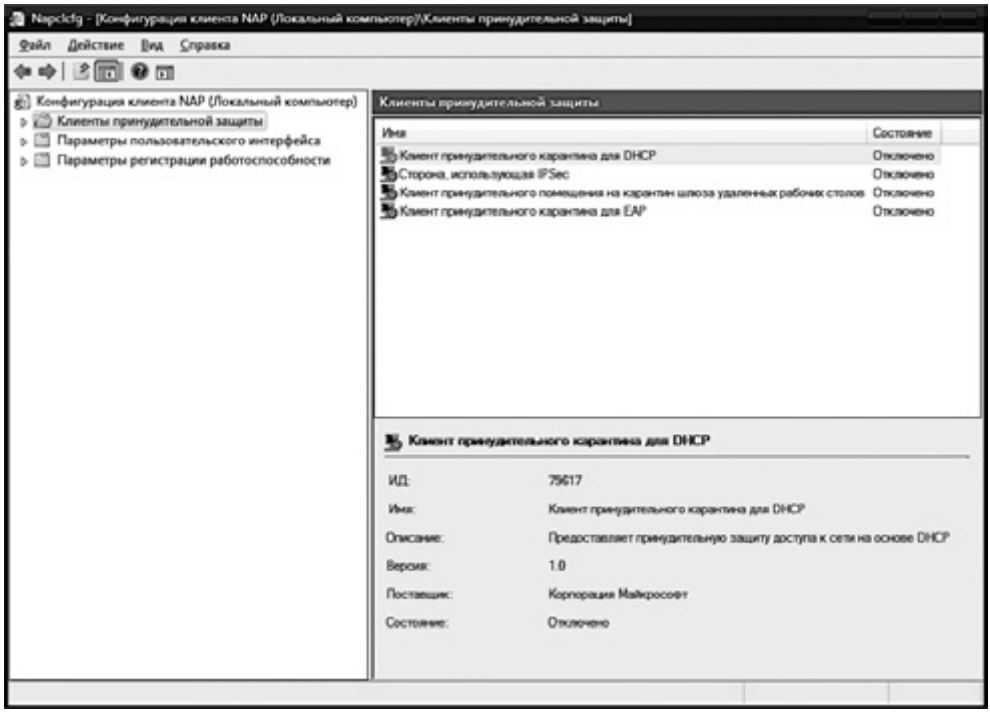


Рис. 7.52. Настройка клиента

Direct Access

В Windows 7 максимального (Ultimate) и корпоративного (Enterprise) выпусков Microsoft добавляет такое постоянно включенное свойство, как виртуальная частная сеть — Virtual Private Network (VPN), — которая также называется непосредственным доступом — DirectAccess (DA). Она позволяет удаленному клиенту Интернета получить доступ к корпоративной сети, основанной на доменной структуре. DA-подключение к корпоративной сети создается при загрузке клиентской системы и длится, пока клиент работает и сохраняет подключение к Интернету. Если сетевые проблемы становятся причиной падения сети, подключение будет автоматически установлено заново, как только будут разрешены сетевые подключения. DA использует службу IPsec, работающую через протокол IPv6, который может быть инкапсулирован в IPv4 с помощью различных механизмов, если у локальной системы нет сквозного IPv6-соединения с частной сетью. Удаленные системы могут даже использовать DA, когда они находятся за брандмауэром, поскольку DA может использовать HTTPS (TCP порт 443) в качестве транспорта (IP-HTTPS).

В отличие от обычных VPN-продуктов, удаленные системы, использующие DA для доступа к корпоративной сети, всегда видны и доступны для управления — так же как если бы машина была непосредственно подсоединена к корпоративной сети. IT-отдел предприятия может управлять удаленными системами путем обновления настроек групповой политики или путем активной доставки

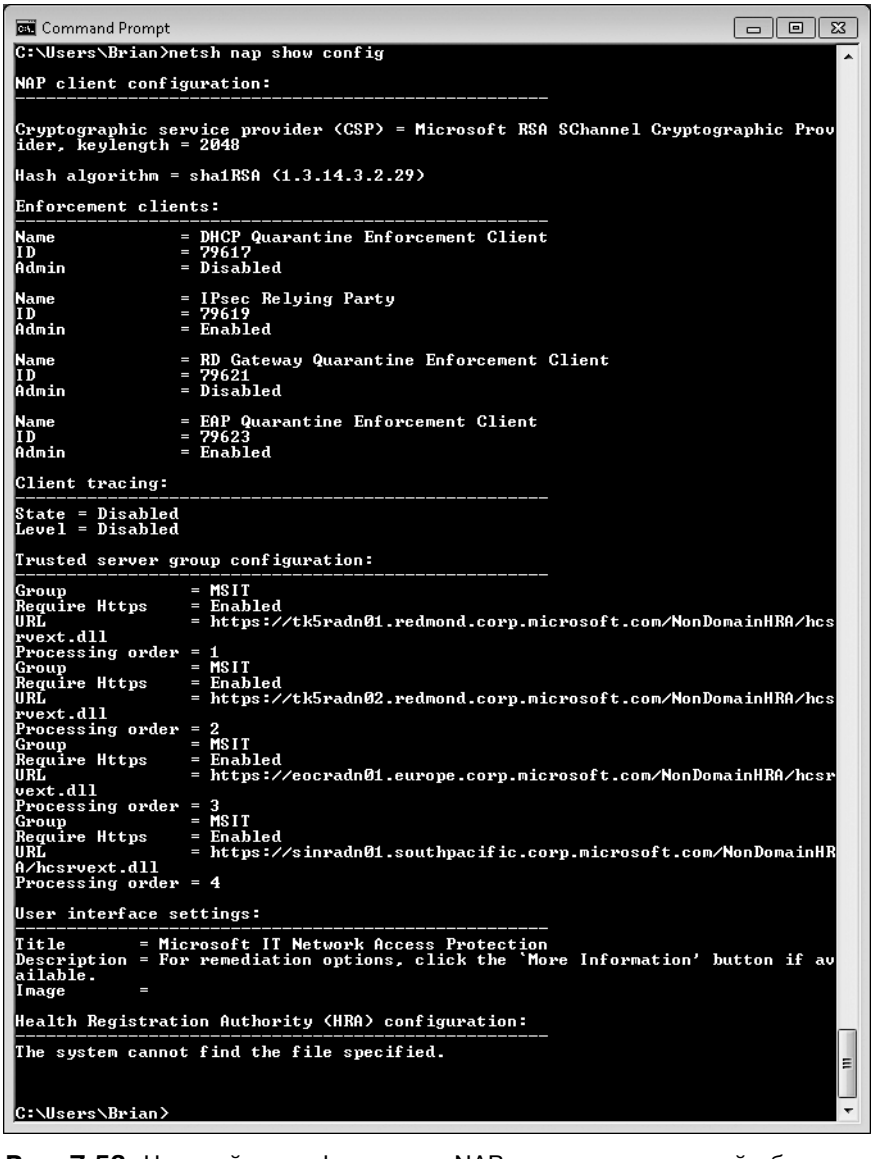


Рис. 7.53. Настройка конфигурации NAP с помощью сетевой оболочки

обновлений при каждом подключении удаленной системы к Интернету. IT-отдел может также указать, какие ресурсы корпоративной сети (приложения, серверы, подсети и т. д.) могут быть доступны пользователю или удаленной системе, подключившейся с помощью DA.

Для повышения уровня безопасности от DA-клиентов может потребоваться рассмотренный в главе 6 механизм обеспечения аутентификации. Для входа в сеть или для ее разблокировки потребуется двухфакторная аутентификация (например, смарткарта или другое аппаратное средство идентификации).

Как показано на рис. 7.54, для подключения DA-клиента к корпоративной сети доступно множество механизмов: IPv6, протокол автоматической внутрисайтовой адресации туннелей — Intra-Site Automatic Tunnel Addressing Protocol (ISATAP), — IPv4 шифрование с IPsec, туннель 6-в-4 tunnel, или Teredo. Во всех случаях устанавливается соединение между удаленным клиентом и DA-сервером. Этот сервер предоставляет защиту отказа в обслуживании — Denial of Service (DoS) — за счет снижения скорости трафика согласования подключения, использующегося для подключения к нему, и он работает в качестве туннельного шлюза IPv6 между удаленным клиентом и корпоративной сетью. Для управления доступом к корпоративной сети DA-сервер также функционирует как шлюз безопасности IPsec на основе IPv6, подобно VPN-серверу или VPN-концентратору клиентского доступа.

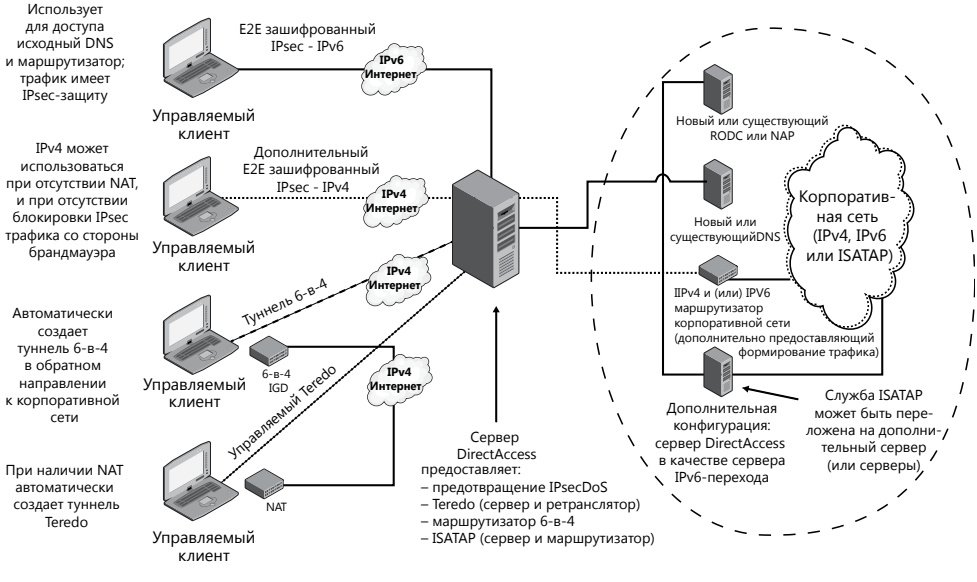


Рис. 7.54. Подключение DA-клиента к корпоративной сети

У клиента, как правило, имеется два IPv6-туннеля к DA-серверу: туннель инфраструктуры и туннель интранета. Туннель инфраструктуры предназначен для связи с серверами корпоративной инфраструктуры, например с сервером Domain Name System (DNS) и с контроллерами домена. Туннель инфраструктуры создается автоматически при начальной загрузке клиента и не требует от пользователя входа в сеть. Туннель интранета создается, когда пользователь входит в сеть, и по нему переносится трафик для пользователя.

DA также работает с NAP. В этом случае сервер Health Registration Authority (HRA) помещается за пределами корпоративного брандмауэра (что часто называют демилитаризованной зоной — DeMilitarized Zone, или DMZ). Клиент настроен с именем HRA (которое может быть разрешено в IP-адрес с помощью общедоступного DNS-сервера). Когда клиент загружает систему, он контактирует с сервером HRA и отправляет свое заявление о состоянии работоспособности —

Statement of Health. Если состояние клиента не удовлетворяет требованиям, он должен обратиться к корректирующим серверам, которые также находятся в DMZ. Если состояние работоспособности клиента соответствует требованиям, он получает сертификат работоспособности, который затем может быть использован с IPsec для подключения к DA-серверу.

Заключение

Сетевая архитектура Windows предоставляет гибкую инфраструктуру для сетевых API-интерфейсов, драйверов сетевых протоколов и драйверов сетевых адаптеров. Сетевая архитектура Windows использует разделение ввода-вывода по уровням, чтобы придать сетевой поддержке возможность для расширения по мере расширения компьютерных сетей. Аналогичным образом, новые API могут стать связующим звеном для существующих драйверов протоколов Windows. И наконец, диапазон сетевых API-интерфейсов, реализованный в Windows, предоставляет разработчикам сетевых приложений широкий спектр возможных реализаций, у каждой из которых могут быть разные модели программирования и поддержка разных протоколов.