Міністерство освіти і науки України
Чернівецький національний університет
імені Юрія Федьковича

Підлягає поверненню на кафедру

Операційні системи

*Методичні рекомендації
до лабораторних робіт*

Чернівці
"*Рута*"
2010

ББК 32.973.26-018.2
О–609
УДК 004.451

Друкується за ухвалою редакційно-видавничої ради
Чернівецького національного університету
імені Юрія Федьковича

Рецензент: Остапов С.Е., доктор фіз.-мат. наук, професор.

О–609 **Операційні** системи: методичні рекомендації до лабораторних робіт / Укл.: Жихаревич В.В. – Чернівці: Рута, 2010. – 248 с.

Запропоноване видання узгоджене з навчальною програмою з відповідної дисципліни та може використовуватися як посібник-довідник для студентів, які вивчають побудову та функціонування операційних систем. Розглядається структура операційної системи Windows, а також функції та робота окремих її підсистем. Достатня увага приділяється питанням керування пам'яттю та процесами в операційній системі, а також структурі файлової системи NTFS та структурі підсистеми захисту.

Для студентів вищих навчальних закладів, які навчаються за напрямком підготовки „Програмна інженерія".

ББК 32.973.26-018.2

УДК 004.451

ЗМІСТ

ВСТУП

Дисципліна „Операційні системи” є однією з головних у напрямках програмної інженерії та комп’ютерних наук. Мета викладання даної дисципліни – набуття студентами базових знань з основ побудови та функціонування сучасних операційних систем, які необхідні у подальшому навчанні, а також у практичній діяльності на виробництві.

Запропонований лабораторний практикум ілюструє визначальні положення лекційного курсу "Операційні системи" на прикладі 32-розрядної версії операційної системи (ОС) Windows (Windows NT, 2000, XP, Vista), розробленої корпорацією Microsoft.

Книга складається з 12 лабораторних робіт, має традиційну побудову та відображає такі теми, як основи побудови ОС Windows, керування процесами і потоками, організація пам’яті, структура файлової системи та безпека. Методичні вказівки містять багато ілюстрацій та прикладів. Суть лабораторних занять полягає у розробці невеликих програм, які ілюструють окремі аспекти реалізації ОС Windows, а також у використанні для цих цілей різноманітних інструментальних засобів.

Успішне виконання лабораторних робіт та засвоєння дисципліни в цілому передбачає попереднє знання архітектури комп’ютерів та основ програмування алгоритмічними мовами C та Pascal. Окрім того, досить важливо ознайомитися зі змістом однієї або кількох монографій з операційних систем, системного програмування, а також мати довідникову інформацію про API-функції ОС Windows.

Методичні вказівки призначені для студентів спеціальності „Програмне забезпечення автоматизованих систем” напрямку підготовки „Програмна інженерія”.

Лабораторна робота № 1
Дослідження особливостей функціонування операційної системи Windows шляхом створення та аналізу роботи Windows-додатків

Теоретична частина

Історія створення операційної системи Windows

Історію розвитку операційних систем корпорації Microsoft можна умовно розділити на три етапи:

- MS-DOS і MS-DOS+Windows 3.1;
- так звані споживчі (consumer) версії Windows (Windows 95/98/Me);
- остання лінія операційних систем, що ведуть свій початок від Windows NT (Windows NT/2000/XP/Vista).

Однозадачна 16-розрядна ОС MS-DOS випущена на початку 80-х років і потім широко застосовувалася на комп'ютерах із процесором x86. Спочатку MS-DOS була досить примітивна (деградація ОС), її оболонка займалася переважно обробкою командного рядка, але в подальші версії було внесено багато поліпшень, запозичених здебільшого з ОС Unix. Потім під впливом успіхів дружнього графічного інтерфейсу корпорації Macintosh була розроблена система Windows. Особливо значного поширення набули версії Windows 3.0, 3.1 і 3.11. Первинно це була не самостійна ОС, а швидше багатозадачна (з невитісняючою багатозадачністю) графічна оболонка MS-DOS, яка контролювала комп'ютер і файлову систему.

У 1995 р. була випущена 32-розрядна ОС Windows 95, де була реалізована витісняюча багатозадачність. ОС Windows 95 включала великий об'єм 16-розрядного коду, головним чином для забезпечення спадкоємності з додатками MS-DOS. 16-розрядний код наявний і в подальших версіях цієї серії Windows 98 і Windows Me. Іншою проблемою даної версії Windows, багато в чому зумовленою тією ж причиною, була нереєнтерабельність істотної частини коду ядра. Так, якщо один із потоків був зайнятий модифікацією даних в ядрі, інший потік, щоб не одержати ці дані в суперечливому стані, змушений був чекати, тобто не міг

скористатися системними сервісами. Це часто зводило нанівець переваги багатозадачності.

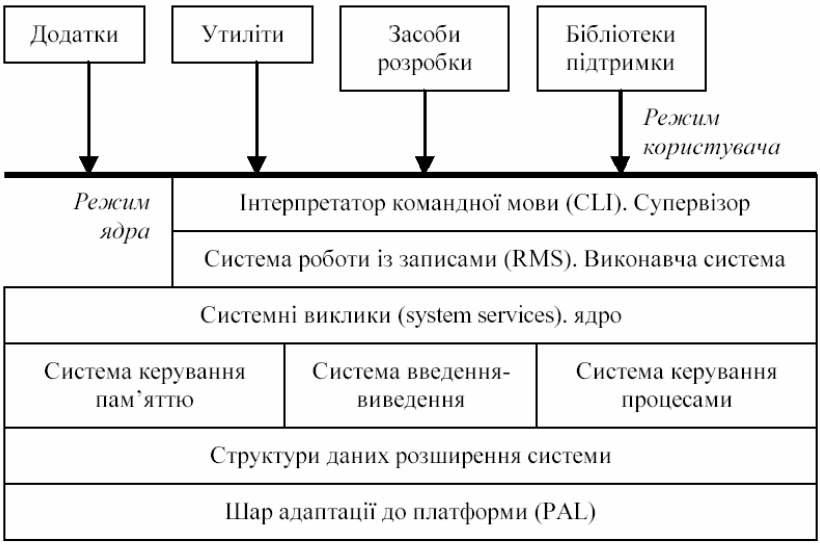
ОС Windows NT (New Technology) – нова 32-розрядна ОС, сумісна з попередніми версіями Windows по інтерфейсу. Роботу над створенням системи очолив Девід Катлер, один із ключових розробників ОС VAX VMS. Ряд ідей системи VMS наявний у NT (див. рис. 1.1). Помітна спадкоємність у системі управління великим адресним простором і резидентною областю процесу, в системі пріоритетів звичайних процесів і процесів реального часу, в засобах синхронізації і т.д. Водночас Windows NT – це абсолютно новий амбітний проект розробки системи з урахуванням новітніх досягнень у області архітектури мікроядра. Перша версія, названа Windows NT 3.1 для відповідності популярної Windows 3.1, була випущена в 1993 р. Комерційного успіху добилася версія Windows NT 4.0, що запозичила графічний інтерфейс Windows 95. На початку 1999 р. була випущена Windows NT 5.0, перейменована в Windows 2000. Наступна версія цієї ОС даної серії – Windows XP з'явилася в 2001 р., а Windows Server 2003 – в 2003 р. В даний час випущена Windows Vista, раніше відома під кодовим ім'ям Longhorn, – нова версія Windows, що продовжує лінійку Windows NT.

Об'єм початкових текстів ядра ОС Windows невідомий. За деякими оцінками, об'єм ядра Windows NT 3.5 складає приблизно 10Мб, а з кожною новою версією ОС Windows цей об'єм неухильно збільшується в півтора-два рази.

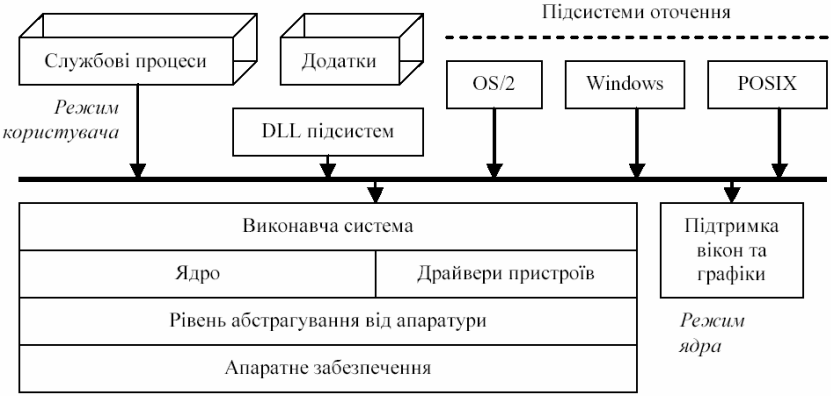
Можливості операційної системи Windows

Перед розробниками системи було поставлене завдання створити операційну систему персонального комп'ютера, призначену для виконання серйозних завдань, а також для домашнього використання. Перелік можливостей системи достатньо широкий, ось лише деякі з них. Операційна система Windows:

- є істинно 32-розрядною, підтримує витісняючу багатозадачність;
- працює на різній апаратній архітектурі і володіє здібністю до порівняно легкого перенесення на нову апаратну архітектуру;
- підтримує роботу з віртуальною пам'яттю;



Структура ОС VAX/VMS



Структура ОС Windows NT

Рис. 1.1. Порівняння архітектури ОС Windows і VAX/VMS

- є повністю реєнтерабельною;
- добре масштабується в системах із симетричною мультипроцесорною обробкою;

- є розподіленою обчислювальною платформою, здатною виступати в ролі як клієнта мережі, так і сервера;
- захищена як від внутрішніх збоїв, так і від зовнішніх деструктивних дій. У додатків немає можливості порушити роботу операційної системи або інших додатків;
- сумісна, тобто її інтерфейс користувача і API сумісні з попередніми версіями Windows і MS-DOS. Вона також уміє взаємодіяти з іншими системами на зразок UNIX OS/2 і NetWare;
- володіє високою продуктивністю незалежно від апаратної платформи;
- забезпечує простоту адаптації до глобального ринку за рахунок підтримки Unicode;
- підтримує багатопоточність і об'єктну модель.

Загальний опис структури операційної системи Windows

Архітектура ОС Windows зазнала ряд змін у процесі еволюції. Перші версії системи мали мікроядерний дизайн, заснований на мікроядрі Mach, яке було розроблене в університеті Карнеги-Меллона. Архітектура пізніших версій системи мікроядерною вже не являється.

Причина полягає в поступовому подоланні основного недоліку мікроядерної архітектури – додаткових накладних витрат, пов'язаних із передачею повідомлень. На думку фахівців Microsoft, чисто мікроядерний дизайн комерційно невигідний, оскільки неефективний. Тому великий об'єм системного коду, в першу чергу управління системними викликами й екранна графіка, був переміщений з адресного простору користувача в простір ядра і працює в привілейованому режимі. В результаті в ядрі ОС Windows переплетені елементи мікроядерної архітектури і елементи монолітного ядра (комбінована система). Сьогодні мікроядро ОС Windows дуже велике (більше 1 Мб), щоб носити префікс "мікро". Основні компоненти ядра Windows NT розташовуються в пам'яті, що витісняється, і взаємодіють один з одним шляхом передачі повідомлень, як і прийнято в мікроядерних операційних системах. Водночас всі компоненти ядра працюють в одному ад-

ресному просторі й активно використовують загальні структури даних, що властиво операційним системам із монолітним ядром.

Висока модульність і гнучкість перших версій Windows NT дозволила успішно перенести систему на такі відмінні від Intel платформи, як Alpha (корпорація DEC), Power PC (IBM) і MIPS (Silicon Graphic). Пізніші версії обмежуються підтримкою архітектури Intel x86.

Спрощена схема архітектури, зорієнтована на виконання Win32-додатків, показана на рис. 1.2.

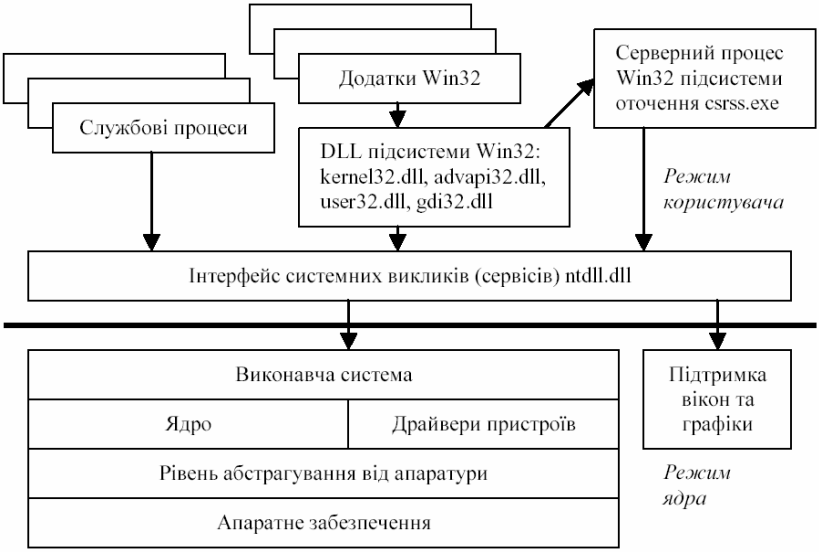


Рис. 1.2. Спрощена архітектурна схема ОС Windows

ОС Windows складається з компонентів, що працюють у режимі ядра, і компонентів, що працюють у режимі користувача. Незважаючи на міграцію системи у бік монолітного ядра, вона зберегла деяку структуру. У вищеподаній схемі виразно видимі кілька функціональних рівнів, кожний з яких користується сервісами нижчого рівня.

Завдання *рівня абстрагування від устаткування (hardware abstraction layer, HAL)* – приховати апаратні відмінності апаратної архітектури для потенційного перенесення системи з однієї

платформи на іншу. HAL надає вищележачим рівням апаратні пристрої в абстрактному вигляді, вільному від індивідуальних особливостей. Це дозволяє ізолювати ядро, драйвери і виконавчу систему ОС Windows від специфіки устаткування (наприклад, від відмінностей між материнськими платами).

Ядром звичайно називають всі компоненти ОС, що працюють у привілейованому режимі роботи процесора або в режимі ядра. Корпорація Microsoft називає *ядром (kernel)* компонент, що знаходиться в невивантажуваній пам'яті і містить низькорівневі функції операційної системи, такі, як диспетчеризація переривань і виключень, планування потоків і ін. Воно також надає набір процедур і базових об'єктів, використовуваних компонентами вищих рівнів.

Ядро і HAL є апаратно-залежними і написані на мовах Сі та асемблера. Верхні рівні написані на мові Сі і є машинно-незалежними.

Виконавча система (executive) забезпечує управління пам'яттю, процесами і потоками, захист, уведення-виведення і взаємодію між процесами. *Драйвери пристроїв* містять апаратно-залежний код і забезпечують трансляцію призначених для користувача викликів у запити, специфічні для конкретних пристроїв. Підсистема підтримки вікон і графіки реалізує функції графічного інтерфейсу користувача (GUI), відоміші як Win32-функції модулів USER і GDI.

У просторі користувача працюють різноманітні сервіси (аналоги демонів в Unix), керовані диспетчером сервісів, що виконують системні завдання. Деякі системні процеси (наприклад, обробка входу в систему) диспетчером сервісів не управляються і називаються фіксованими процесами підтримки системи. Призначені для користувача додатки (user applications) бувають п'яти типів: Win32, Windows 3.1, MS-DOS, POSIX і OS/2 1.2. Середовище для виконання процесів користувача надають три підсистеми оточення: Win32, POSIX і OS/2. Таким чином, додатки користувача не можуть викликати системні виклики ОС Windows безпосередньо, а змушені звертатися до DLL (динамічно зв'язаних бібліотек) підсистем.

Основні компоненти ОС Windows реалізовані в таких системних файлах, що знаходяться в каталозі system32:

- ntoskrnl.exe – виконавча система і ядро;
- ntdll.dll – внутрішні функції підтримки і інтерфейси диспетчера системних сервісів з функціями виконавчої системи;
- hal.dll – рівень абстрагування від устаткування;
- win32k.sys – частина підсистеми Win32, що працює в режимі ядра;
- kernel32.dll, advapi32.dll, user32.dll, gdi32.dll – основні dll підсистеми Win32.

Підсистема Win32

Оскільки практична частина даної лабораторної роботи припускає розробку і виконання різноманітних Win32-додатків, які працюють у середовищі, що створюється Win32-підсистемою, необхідно розглянути її детальніше. Взаємодія між додатком і операційною системою здійснюється за допомогою системних викликів (системних сервісів у термінології Microsoft). Проте додаток не може здійснювати системний виклик безпосередньо (більше того, системні виклики не документовані). Натомість додаток повинен скористатися програмним інтерфейсом ОС – Win32 API.

Win32 API (Application Programming Interface) – основний інтерфейс програмування в сімействі операційних систем Microsoft Windows. Функції Win32 API, наприклад, *CreateProcess* або *CreateFile*, – документовані підпрограми, які можуть викликатися і реалізуються Win32 підсистемою.

До складу Win32 підсистеми (див. рис. 1.2) входять: серверний процес підсистеми оточення csrss.exe, драйвер режиму ядра Win32k.sys, dll-модулі підсистем (kernel32.dll, advapi32.dll, user32.dll і gdi32.dll), що експортують Win32-функції і драйвери графічних пристроїв. У процесі еволюції структура підсистеми зазнала зміни. Наприклад, функції вікон і малювання з метою підвищення продуктивності були перенесені з серверного процесу, що працює в режимі користувача, в драйвер режиму ядра Win32k.sys. Проте ці та подібні зміни ніяк не відбилися на праце-

здатності додатків, оскільки існуючі виклики Win32 API не змінюються з новими випусками системи Windows, хоча їх склад постійно поповнюється.

Додаток, зорієнтований на використання Win32 API, може працювати практично на всіх версіях Windows, незважаючи на те, що самі системні виклики в різних системах різні (див. рис. 1.3). Таким шляхом корпорація Microsoft забезпечує спадкоємність своїх операційних систем.

Рис. 1.3. Підтримка єдиного програмного інтерфейсу для різних версій Windows

При запуску процесу всі необхідні динамічні бібліотеки відображаються на його віртуальний адресний простір, а для швидкого виклику бібліотечної процедури використовується спеціальний вектор передачі.

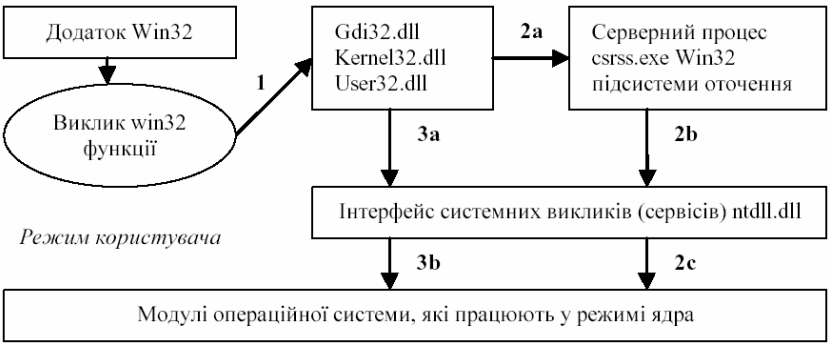


Рис. 1.4. Різні маршрути виконання викликів Win32 API

При виклику додатком однієї з Win32-функцій dll-підсистем може виникнути одна з трьох ситуацій (див. рис. 1.4):

- функція повністю виконується всередині даною dll (крок 1);

- для виконання функції задіюється сервер csrss, для чого йому посилається повідомлення (крок 2a, за яким зазвичай ідуть кроки 2b і 2c);
- даний виклик транслюється в системний сервіс (системний виклик), який зазвичай обробляється в модулі ntdll.dll (кроки 3a і 3b). Наприклад Win32-функція ReadFile виконується за допомогою недокументованого сервісу *NtReadFile*.

Деякі функції (наприклад, CreateProcess) вимагають виконання обох останніх пунктів.

У перших версіях ОС Windows практично всі виклики Win32 API виконувалися, дотримуючись маршруту 2 (2a, 2b, 2c). Після того, як істотна частина коду системи для збільшення продуктивності була перенесена в ядро (починаючи з Windows NT 4.0), виклики Win32 API, як правило, йдуть безпосередньо по 3-му (3a, 3b) шляху, минувши підсистему оточення Win32. У даний час лише невелика кількість викликів виконується по довгому 2-му маршруту.

Окрім перерахованих, найбільш важливих dll-бібліотек, у системному каталозі system32 є велика кількість інших dll-файлів. У даний час кількість викликів API складає кілька десятків тисяч.

Особливості створення Windows-додатків

Розглянемо коротко відмінності ОС MS-DOS та Windows. Операційні системи MS-DOS і Windows підтримують дві абсолютно різні ідеології програмування. Програма DOS після свого запуску повинна бути постійно активною. Якщо їй що-небудь необхідно, наприклад, отримати чергову порцію даних із пристрою введення-виведення, то вона сама повинна виконувати відповідні запити до операційної системи. При цьому програма DOS працює за певним алгоритмом, вона завжди знає, що і коли їй необхідно робити. У Windows все навпаки. Програма пасивна. Після запуску вона очікує, коли їй приділить увагу операційна система. Операційна система робить це посиланням спеціально оформлених груп даних, які називаються *повідомленнями* (*Messages*). Повідомлення можуть бути різного типу, вони функціонують у системі достатньо хаотично, і програмний додаток не знає, якого типу повідомлення прийде наступним. Звідси випливає, що логіка по-

будови Windows-додатка повинна забезпечувати коректну і передбачувану роботу при поступленні повідомлень будь-якого типу. Тут можна навести аналогію між механізмом повідомлень Windows і механізмом переривань в архітектурі IBM PC. Для забезпечення нормального функціонування своєї програми програміст повинен вміти ефективно використовувати функції інтерфейсу прикладного програмування API (Application Program Interface) операційної системи.

Windows підтримує два типи програмних додатків:

- *віконний додаток* – будується на базі спеціального набору функцій (API), які складають графічний інтерфейс користувача GUI (Graphic User Interface). Віконний додаток являє собою програму, яка виведення на екран виконує в графічному вигляді. Першим результатом роботи віконного додатка є відображення на екрані спеціального об'єкта – вікна. Після того як вікно відобразиться на екрані, вся робота додатка спрямована на те, щоб підтримувати його в актуальному стані;

- *невіконний додаток*, який також називають *консольним*, являє собою програму, що працює в текстовому режимі. Робота консольного додатка нагадує роботу програми MS-DOS. Але це лише зовнішнє враження. Консольний додаток забезпечується спеціальними функціями Windows.

Різниця між двома типами додатків Windows визначається тим, з яким типом інформації вони працюють. Основний тип додатків у Windows – віконний, тому на прикладі їх розробки ми почнемо знайомство з цією операційною системою.

Будь-який віконний Windows-додаток має типову структуру, основу якої складає так званий *каркасний додаток*. Цей додаток містить мінімально необхідний програмний код для забезпечення функціонування повноцінного Windows-додатка. Дослідження роботи каркасних додатків відображає основні особливості взаємодії програм з операційною системою Windows. Більше того, написавши та налагодивши один раз каркасний додаток, ви можете використовувати його в подальшому, як основу для написання будь-якого іншого, значно більш складнішого Windows-додатка.

Мінімальний каркасний додаток Windows складається з трьох частин:

- головної функції;
- циклу обробки повідомлень;
- віконної функції.

Виконання будь-якого віконного Windows-додатка починається з *головної функції*. Вона містить код, що здійснює налаштування (ініціалізацію) додатка в середовищі операційної системи Windows. Видимим для користувача результатом роботи головної функції є поява на екрані графічного об'єкта у вигляді вікна. Останньою дією коду головної функції є створення *циклу обробки повідомлень*. Після його створення додаток стає пасивним і починає взаємодіяти з оточуючим середовищем за допомогою спеціальним чином оформлених даних – *повідомлень*. Обробка повідомлень, які надходять у додаток, здійснюється спеціальною функцією, яка називається *віконною*. Віконна функція унікальна тим, що може бути викликана тільки з операційної системи, а не з додатка, який її містить. Тіло віконної функції має певну структуру. Таким чином, Windows-додаток повинен складатися принаймні з трьох перелічених елементів. Розглянемо приклад Windows-додатка, написаного мовою програмування Pascal.

Приклад 1.1

```
uses Windows,Messages;
// Функція WindowProc викликається операційною системою
// Windows та отримує як параметри повідомлення з черги
// повідомлень даного додатка
function WindowProc conv arg_stdcall (
Window: HWND; //дескриптор (вказівник або заголовок) вікна
Mess: UINT;         //ідентифікатор повідомлень
Wp: WParam;         //перший параметр повідомлення
Lp: LParam          //другий параметр повідомлення
): LRESULT;
begin  case Mess of
WM_DESTROY: begin  //повідомлення для завершення роботи
PostQuitMessage(0);      //очистка Windows-специфічних об'єктів
Result := 0; end; else
```

```
// в іншому випадку сюди потрапляють усі повідомлення, які не
// обробляються у даній віконній функції.
//Далі ці повідомлення направляються назад у Windows:
Result := DefWindowProc(Window, Mess, Wp, Lp);end;end;
var                    //параметри основної програми:
wc: TWndClass; //структура класу вікна
wnd: HWnd;      //дескриптор вікна
Msg: TMsg;       //структура повідомлення
begin                              //початок – визначення класу вікна
FillChar(wc, SizeOf(wc), 0);//довжина структури TWndClass
with wc do begin
style:=CS_HREDRAW + CS_VREDRAW;//стиль класу вікна
lpfnWndProc := @WindowProc;  //адреса функції вікна
cbClsExtra := 0;                    //для внутрішнього використання
cbWndExtra := 0;                    //для внутрішнього використання
hInstance := System.hInstance;     //дескриптор даного додатка
hIcon := LoadIcon(THandle(NIL), IDI_APPLICATION);// іконка
hCursor := LoadCursor(THandle(NIL), IDC_ARROW);  //курсор
hbrBackGround := GetStockObject(WHITE_BRUSH);   //білий колір
lpszMenuName := nil;                                    //без меню
lpszClassName := 'Демонстраційна програма';     //ім'я класу вікна
end;
if RegisterClass(wc) = 0 then Exit;   //реєстрація класу вікна
  //створити вікно та присвоїти дескриптор вікна змінній wnd
wnd := CreateWindow(
wc.lpszClassName,                  //ім'я класу вікна
'Каркас програми для Windows',//заголовок вікна
WS_OVERLAPPEDWINDOW,  //стиль вікна
CW_USEDEFAULT, //X-координата верхнього лівого кута вікна
CW_USEDEFAULT, //Y-координата верхнього лівого кута вікна
CW_USEDEFAULT,  //ширина вікна
CW_USEDEFAULT,  //висота вікна
0,                          //дескриптор батьківського вікна
0,                         //дескриптор меню вікна
HInstance,                //ідентифікатор додатка, який створює вікно
NIL);                      //вказівник на область даних додатка
```

```
//показати вікно та перемалювати вміст
ShowWindow(wnd, SW_RESTORE);
UpdateWindow(wnd);
//запустити цикл обробки повідомлень
while GetMessage(Msg,0,0,0) do begin
TranslateMessage(Msg);      //дозволити використання клавіатури
DispatchMessage(Msg);       //повернути керування Windows
end; end.
```

Розберемо більш детально суть дій, які виконуються кожним із трьох елементів Windows-додатка. З тексту програми, наведеного вище, видно, що мінімальний Windows-додаток складається з двох частин: основної програми (головної функції) та віконної функції WindowProc. Мета головної функції – повідомити системі щодо нового для неї додатка, його властивостях та особливостях. Для досягнення цієї мети головна функція виконує такі дії:

- визначає та реєструє клас вікна додатка;
- створює та відображає вікно додатку зареєстрованого класу;
- створює та запускає цикл обробки повідомлень для додатка;
- завершує програму при отриманні віконною функцією відповідного повідомлення.

Віконна функція отримує всі повідомлення, призначені даному вікну, та обробляє їх, маючи можливість виклику для цього інших функцій.

Видима частина роботи каркасного додатка полягає у створенні нового вікна на екрані. Воно відповідає всім вимогам стандартного вікна додатка Windows, тобто його можна розгорнути, згорнути, змінити розмір, перемістити в інше місце екрана і т.д. Для цього, як ви бачите, непотрібно дописувати ані рядка додаткового програмного коду, а всього лише задовольнити певні вимоги, що висуваються до додатків із боку операційної системи.

Цикл обробки повідомлень Windows-додатків

Повідомлення в Win32 являє собою об’єкт особливої структури, що формується Windows. Формування та забезпечення доставки цього об’єкта у необхідне місце в системі дозволяють керувати як роботою самої системи Windows, так і роботою завантажених Windows-додатків. Ініціювати формування повідомлення мо-

же кілька джерел: користувач, самий додаток, система Windows, інші додатки. Саме наявність механізму повідомлень дозволяє Windows реалізувати багатозадачність, яка при роботі на одному процесорі є, звичайно ж, псевдомультизадачністю. Windows підтримує чергу повідомлень для кожного додатка. Запуск додатка автоматично передбачає формування для нього власної черги повідомлень, навіть якщо цей додаток і не буде нею використовуватися. Останнє малоймовірно, оскільки в цьому випадку в додатка не буде зв'язку з навколишнім середовищем, і воно являтиме собою „річ у собі".

Формат всіх повідомлень Windows однаковий та описується структурою, шаблон якої на мові Pascal має вигляд:

```
TMsg =  packed record
  hwnd: HWND;
  message: UINT;
  wParam: WPARAM;
  lParam: LPARAM;
  time: DWORD;
  pt: TPoint;       end;
```

Поле hwnd структури повідомлення містить значення дескриптора вікна, якому призначене повідомлення. В полі message операційна система Windows розміщує 32-бітну константу – ідентифікатор повідомлення. Для зручності всі ці константи мають символічні імена, які починаються з WM_ (Window Message). Поля wParam та lParam призначені для того, щоб Windows могла розмістити в них додаткову інформацію щодо повідомлення, необхідну для його правильної обробки. Ці поля, наприклад, використовуються при обробці повідомлень щодо вибору пунктів меню або щодо натискання клавіш клавіатури. В полі time система Windows записує інформацію про час, коли повідомлення було розміщено у чергу повідомлень. Поле pt містить координати курсора миші в момент розміщення повідомлень у чергу та являє собою структуру вигляду

```
TPoint = record
   x: Longint;
   y: Longint;  end;
```

Таким чином, якщо в системі відбулася яка-небудь подія, наприклад, деякому додаткові необхідно перемалювати своє вікно, в результаті чого Windows сформувала повідомлення WM_PAINT. Дане повідомлення потрапляє в чергу повідомлень додатка, який створив вікно. Для того щоб додаток міг обробляти це (або будь-яке інше) повідомлення, необхідно спочатку виявити його в черзі повідомлень. Із цією метою, після відображення вікна на екрані, програма „входить" у спеціальний цикл, який називається *циклом обробки повідомлень*. Вийти з цього циклу можна лише з приходом повідомлення WM_QUIT. У цьому випадку функція GetMessage повертає нульове значення. У випадку приходу інших повідомлень функція GetMessage повертає ненульове значення, в результаті чого і здійснюється вхід безпосередньо в тіло циклу обробки повідомлення.

Функція GetMessage виконує такі дії:

- постійно переглядає чергу повідомлень;
- вибирає повідомлення, які задовольняють задані у функції параметри;
- заносить інформацію про повідомлення в екземпляр структури Msg;
- передає керування в цикл обробки повідомлень.

Цикл обробки повідомлень складається всього з двох функцій: TranslateMessage та DispatchMessage. Ці функції мають єдиний парметр – вказівник на екземпляр структури Msg, попередньо заповнений інформацією про повідомлення функцією GetMessage.

Функція TranslateMessage призначена для виявлення повідомлень від клавіатури для даного додатка. Якщо додаток не буде самостійно обробляти введення з клавіатури, то ці повідомлення передаються для обробки назад у Windows.

Функція DispatchMessage призначена для передачі повідомлення віконній функції. Така передача виконується не напряму, оскільки сама функція DispatchMessage нічого не знає про розташування віконної функції, а опосередковано – за допомогою системи Windows. При цьому:

- функція DispatchMessage повертає повідомлення операційній системі;
- Windows, використовуючи опис класу вікна, передає повідомлення відповідній віконній функції додатка;
- після обробки повідомлення віконною функцією керування повертається операційній системі;
- Windows передає керування функції DispatchMessage;
- DispatchMessage завершує своє виконання.

Оскільки виклик функції DispatchMessage останній у циклі, то керування знову передається функції GetMessage. Функція GetMessage вибирає чергове повідомлення з черги повідомлень і, якщо воно задовольняє параметрам, заданим при виклику цієї функції, то виконується тіло циклу. Цикл обробки повідомлень виконується доти, поки не прийде повідомлення WM_QUIT. Отримання цього повідомлення – єдина умова, при якій програма може вийти з циклу обробки повідомлень.

Обробка повідомлень у віконній функції Windows-додатків

Віконна функція призначена для організації адекватної реакції з боку Winows-додатка на дії користувача і підтримки в актуальному стані того вікна додатку, повідомлення якого вона обробляє. Додаток може мати кілька віконних функцій. Їх кількість визначається кількістю класів вікон, зареєстрованих у системі. Коли для вікна Windows-додатка з'являється повідомлення, операційна система Windows виконує виклик відповідної віконної функції. Повідомлення, в залежності від джерела їх появи у віконній функції, можуть бути двох типів: синхронні та асинхронні. До *синхронних* повідомлень відносять ті повідомлення, які розміщуються у чергу повідомлень додатка і очікують моменту, коли вони будуть вибрані функцією GetMessage. Після цього обрані повідомлення потрапляють у віконну функцію, де і виконується їх обробка. *Асинхронні* повідомлення попадають у віконну функцію в екстреному порядку, оминаючи при цьому всі черги. Асинхронні повідомлення, зокрема, ініціюються деякими функціями Win32 API, такими як CreateWindow або UpdateWindow. Координацію синхронних і асинхронних повідомлень здійснює Windows. Якщо розглядати синхронне повідомлення, то його обрання виконуєть-

ся функцією GetMessage з наступною передачею його назад у Windows функцією DispatchMessage. Асинхронне повідомлення, незалежно від джерела, який ініціює його появу, спочатку потрапляє у Windows і потім – у необхідну віконну функцію.

У схемі, реалізованій у Windows, обробка повідомлень додатком виконується у два етапи: на першому етапі додаток вибирає повідомлення з черги та відправляє його назад у внутрішні структури Windows; на другому етапі Windows викликає необхідну віконну функцію додатка, передаючи їй параметри повідомлення. Перевага цієї схеми в тому, що Windows самостійно вирішує всі питання організації ефективної роботи додатків.

Отже, при надходженні повідомлення Windows викликає віконну функцію і передає їй ряд параметрів. Усі вони беруться з відповідних полів повідомлення. Центральним місцем віконної функції є синтаксична конструкція, в задачу якої входить розпізнавання отриманого повідомлення за його типом (параметр Mess) і передача керування на ту гілку коду віконної функції, яка продовжує подальшу роботу з параметрами повідомлення. Для цього використовується оператор case (див. приклад 1.1).

Практична частина

1. Складіть програму, наведену у прикладі 1.1 теоретичної частини даної лабораторної роботи. Запустіть програму та переконайтеся, що вона працює правильно (повинно з'явитися вікно Windows-додатка).

2. Оформіть звіт із виконаної лабораторної роботи, який повинен містити текст програми, та дайте відповіді на контрольні запитання.

Контрольні запитання та завдання

1. Які можливості операційної системи Windows?
2. Опишіть загальну структуру операційної системи Windows.
3. Що являють собою API-функції?
4. Поясніть відмінності функціонування програм у операційних системах MS-DOS та Windows.
5. Опишіть структуру мінімального Windows-додатка.
6. Що являє собою повідомлення? Опишіть процес обробки повідомлень у віконній функції Windows-додатків.

Лабораторна робота № 2
Дослідження особливостей організації ресурсів інтерфейсу користувача для Windows-додатків

Теоретична частина

Ресурси Windows-додатків

Для включення ресурсів у Windows-додатки використовується загальноприйнята технологія. *Ресурс* – це спеціальний об'єкт, який використовується програмою, але не визначений в її тілі. До ресурсів належать такі елементи інтерфейсу користувача: значки, меню, вікна діалогу, растрові зображення тощо.

Визначення ресурсів виконується у текстовому файлі з розширенням .rc, в якому для опису кожного типу ресурсу використовуються спеціальні оператори. Підготовку цього файла можна вести двома способами: ручним та автоматизованим.

Ручний спосіб передбачає таке:

- розробник ресурсу добре знає оператори, необхідні для опису конкретного ресурсу;
- введення тексту ресурсного файла виконується за допомогою редактора, який не вставляє у текст, що вводиться елементи форматування.

Автоматичний спосіб створення ресурсного файла передбачає використання спеціальної програми – редактора ресурсів, який дозволяє візуалізувати процес створення ресурсу. Кінцеві результати роботи цієї програми можуть бути двох видів: у вигляді текстового файла з розширенням .rc, який у подальшому можна редагувати вручну, або у вигляді двійкового файла, вже придатного для включення у виконавчий файл додатка.

Припустимо, що одним із цих способів отримуємо файл ресурсів у текстовому вигляді. Після того як ресурси описані у файлі з розширенням .rc, вони повинні бути перетворені у вигляд, придатний для включення їх у спільний виконавчий файл додатка. Для цього необхідно:

- скомпілювати ресурсний файл; на цьому кроці виконується перетворення текстового зображення ресурсного фала з розширенням .rc у двійкове зображення з розширенням .res;

- включити ресурси у виконавчий файл додатка.

Далі на конкретних прикладах ми розглянемо порядок виконання цих кроків для включення деяких типів ресурсів у Windows-додатки.

Меню у Windows-додатках

Меню в системі Windows є найбільш розповсюдженим елементом інтерфейсу користувача. Для того щоб включити меню в додаток, необхідно реалізувати нижчепоказану послідовність кроків.

1. Розробити сценарій меню. Перш ніж приступити до процесу включення меню в конкретний додаток, розробимо його логічну схему. Цей крок необхідний для того, щоб вже на стадії проектування забезпечити ергономічні властивості додатка. Адже меню – це один із небагатьох елементів інтерфейсу, з яким користувач вашого додатка буде постійно мати справу. Тому схема меню повинна мати наглядну ієрархічну структуру, з логічно зв'язаними між собою пунктами цієї ієрархії, що допоможе користувачу ефективно використовувати всі можливості додатка. Для того щоб вести предметну розмову, поставимо перед собою задачу розробити для вікна нашого додатка головне меню. При цьому ми дослідимо можливості виводу у вікно додатка тексту і графіки, а також покажемо способи розв'язання загальних проблем, пов'язаних із розробкою додатка. Наше меню буде досить простим і складатиметься з трьох елементів: „Текст", „Графіка" та „Інформація". Перші два з цих пунктів меню мають укладені спливаючі меню. Ієрархічну структуру меню подано на рис. 2.1.

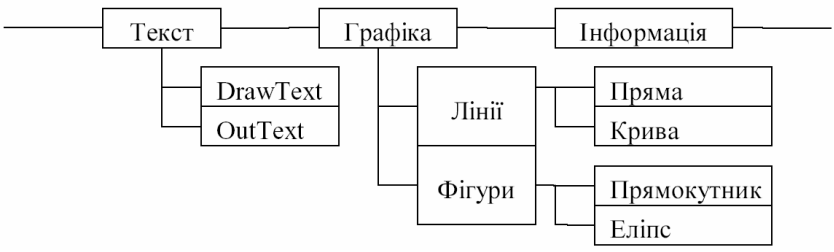


Рис. 2.1. Ієрархічна структура меню

2. Описати схему меню у файлі ресурсів. Для виконання цього опису використовуються спеціальні оператори. В нашому випадку файл ресурсів буде виглядати так:

```
999 MENU DISCARDABLE
  BEGIN
  POPUP "&Текст"
      BEGIN
      MENUITEM "&DrawText", 100 //IDM_DRAWTEXT
      MENUITEM "&TextOut",  101 //IDM_TEXTOUT
      END
  POPUP "&Графіка"
      BEGIN
      POPUP "&Лінії"
          BEGIN
          MENUITEM "&Пряма", 102 //IDM_LINE
          MENUITEM "&Крива", 103 //IDM_CURVE
          END
      POPUP "&Фігури"
          BEGIN
          MENUITEM "&Прямокутник", 104 //IDM_RECTANGLE
          MENUITEM "&Еліпс",       105 //IDM_ELLIPSE
          END
      END
  MENUITEM "&Інформація", 106 //IDM_INFO
  END
```

3. Скомпілювати ресурсний файл. Якщо компіляція відбувається нормально, то створюється файл із розширенням .res. У випадку, якщо компілятор виявляє помилки в початковому ресурсному файлі, то він видає на екран відповідне діагностичне повідомлення.

4. Приєднати файл ресурсів у програмний текст додатка та призначити ідентифікаторам пунктів меню відповідні константи із файла ресурсів, які в подальшому будуть передаватися у віконну процедуру в молодшому слові параметра wParam повідомлення WM_COMMAND:

```
uses Windows, Messages;
{$r menu.res}
const
  MY_MENU       = 999;
  IDM_DRAWTEXT  = 100;
  IDM_TEXTOUT   = 101;
  IDM_LINE      = 102;
  IDM_CURVE     = 103;
  IDM_RECTANGLE = 104;
  IDM_ELLIPSE   = 105;
  IDM_INFO      = 106;
... ... ...
```

5. Підключити меню до того вікна додатка, для роботи з яким воно буде використовуватися. Для цього цілком достатньо на стадії реєстрації вікна помістити в поле lpszMenuName екземпляра структури TwndClass вказівник на поле, який містить ім'я меню:

```
lpszMenuName := MAKEINTRESOURCE(MY_MENU);
```

Можна також виконати такі два кроки:

1) оголосити нову змінну головної функції додатка Menu: hMenu;

2) перед виведенням вікна на екран (ShowWindow) приєднати до нього меню:

```
Menu := LoadMenu(hInstance,
                                MAKEINTRESOURCE(MY_MENU));
SetMenu(wnd, Menu);
```

Після внесення всіх змін у верхній частині вікна додатка з'явиться область меню. Далі необхідно у віконну функцію включити команди, які будуть являти собою реакцію на вибір пунктів цього меню. Ця інформація передається у молодшому слові поля wParam повідомлення WM_COMMAND. У прикладі 2.1 наведено змінений текст каркасного додатка, доповнений меню.

Приклад 2.1

```
uses Windows,Messages;
{$r menu.res}
const
  MY_MENU       = 999;
  IDM_DRAWTEXT  = 100;
  IDM_TEXTOUT   = 101;
  IDM_LINE      = 102;
  IDM_CURVE     = 103;
  IDM_RECTANGLE = 104;
  IDM_ELLIPSE   = 105;
  IDM_INFO      = 106;
var DC: hDC;
    Rect: TRect;
    Rgn: HRgn;
    H_Pen:Hpen;
    d_t:string:='Текст виведено функцією DrawText';
    t_o:string:='Текст виведено функцією TextOut';
    p:array[1..4]of TPoint;
function WindowProc conv arg_stdcall
(Window:HWND;Mess:UINT;Wp:WParam;Lp:LParam):LRESULT;
begin
  case Mess of
  WM_DESTROY: begin
              PostQuitMessage(0);
              Result := 0; end;
  WM_COMMAND:
      begin
        case LoWord(Wp) of
          IDM_DRAWTEXT:
            begin     DC := GetDC(Window);
            GetClientRect(Window,Rect);
            DrawText(DC,@d_t+1,length(d_t),Rect,DT_CENTER);
            ReleaseDC(Window,DC);  end;
          IDM_TEXTOUT:
            begin     DC := GetDC(Window);
```

```
  SelectObject(DC,GetStockObject(ANSI_VAR_FONT));
  SetTextColor(DC,RGB(0,0,255));
  TextOut(DC,10,150,@t_o+1,length(t_o));
  ReleaseDC(Window,DC);  end;
 IDM_LINE:
  begin    DC := GetDC(Window);
  MoveToEx(DC,100,100,nil);
  H_Pen:=CreatePen(PS_SOLID,5,RGB(200,200,200));
  SelectObject(DC,H_Pen);
  LineTo(DC,250,140);
  DeleteObject(H_Pen);
  ReleaseDC(Window,DC);   end;
 IDM_CURVE:
  begin    DC := GetDC(Window);
  p[1].x:=10;p[2].x:=50;p[3].x:=200;p[4].x:=300;
  p[1].y:=200;p[2].y:=100; p[3].y:=350;p[4].y:=100;
  H_Pen:=CreatePen(PS_SOLID,5,RGB(150,150,150));
  SelectObject(DC,H_Pen);
  PolyBezier(DC,p,4);
  DeleteObject(H_Pen);
  ReleaseDC(Window,DC);  end;
 IDM_RECTANGLE:
  begin   DC := GetDC(Window);
  rect.left:=10;rect.right:=60;
  rect.top:=100;rect.bottom:=150;
  FillRect(DC,rect,CreateSolidBrush(RGB(255,0,0)));
  ReleaseDC(Window,DC);  end;
 IDM_ELLIPSE:
  begin   DC := GetDC(Window);
  Rgn:=CreateEllipticRgn(20,20,300,100);
  FillRgn(DC,Rgn,CreateSolidBrush(RGB(0,255,0)));
  ReleaseDC(Window,DC);  end;
 IDM_INFO:
  begin
  MessageBox(Window,'Лабораторна робота з курсу "Операційні системи"', 'Інформаційне повідомлення',MB_OK);
```

```
            end;
          end;
        end;
    else Result := DefWindowProc(Window, Mess, Wp, Lp);
    end;
end;
var wc: TWndClass;wnd:HWnd;Msg: TMsg;Menu: hMenu;
begin
    FillChar(wc, SizeOf(wc), 0);
    with wc do begin
      style:=CS_HREDRAW + CS_VREDRAW;
      lpfnWndProc := @WindowProc;
      cbClsExtra := 0;
      cbWndExtra := 0;
      hInstance := System.hInstance;
      hIcon := LoadIcon(THandle(NIL), IDI_APPLICATION);
      hCursor := LoadCursor(THandle(NIL), IDC_ARROW);
      hbrBackGround := GetStockObject(WHITE_BRUSH);
      lpszMenuName := MAKEINTRESOURCE(MY_MENU);
      lpszClassName := 'Демонстраційна програма';
            end;
    if RegisterClass(wc) = 0 then Exit;
wnd := CreateWindow(wc.lpszClassName,'Програма дослідження
меню', WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, 0,
0, HInstance, nil);
    ShowWindow(wnd, SW_RESTORE);
    UpdateWindow(wnd);
    while GetMessage(Msg,0,0,0) do begin
      TranslateMessage(Msg);
      DispatchMessage(Msg);
    end;end.
```

Скомпілювавши текст даної програми та запустивши виконавчий файл, ви побачите, що реакція програми на вибір пунктів меню полягає у виведенні у вікно додатка відповідних графічних примітивів, тексту та інформаційного повідомлення.

Розглянемо тепер ще одну ключову проблему програмування для Windows – *перемальовування зображення*. Для того щоб зрозуміти її зміст та важливість, виконайте згортання та розгортання вікна додатка або змініть його розмір. Ви побачите, що текст або графіка зникли. Причина цієї ситуації в тому, що Windows не зберігає вмісту вікна, і здійснювати його відновлення після різноманітних із ним дій повинен додаток, який створив це вікно. Windows лише посилає додатку повідомлення WM_PAINT у випадках, коли з вікном були виконані деякі дії, наприклад згортання/розгортання вікна, зміна його розмірів тощо. Додаток, отримавши повідомлення WM_PAINT, як реакцію на нього повинен оновити в необхідній мірі вміст свого вікна. Існує кілька підходів до розв'язання проблеми перемальовування. Найбільш загальний та широко використовуваний спосіб перемальовування зображення ґрунтується на понятті *віртуального* вікна. Розглянемо його.

Перемальовування зображення

В загальному випадку за вміст вікон на екрані відповідають віконні процедури тих додатків, яким ці вікна належать. Роль Windows у цьому процесі мінімальна і полягає в тому, що при певних діях із вікном віконній процедурі, яка відповідає за зв'язок із цим вікном, посилається повідомлення WM_PAINT. Отримавши це повідомлення, віконна процедура повинна вміти заново перемалювати вміст усього вікна або його частини.

Суть перемальовування зображення на основі віртуального вікна полягає у використанні додатком деякої області пам'яті для направлення в нього всього виведення програми. Реальний вивід у вікно додатка здійснюється лише як реакція на отримання повідомлення WM_PAINT. Окрім цього, програма, використовуючи, наприклад, функцію InvalidateRect, може сама собі послати повідомлення WM_PAINT. Додаток робить це, коли йому необхідно вивести нову порцію зображення у вікно додатка. Поняття віртуального вікна настільки важливе для організації роботи Windows, що Win32 API містить ряд функцій, які підтримують роботу з цим вікном: CreateCompatibleDC, SelectObject, GetStockObject, BitBlt, CreateCompatibleBitmap та PatBlt.

StockObject, BitBlt, CreateCompatibleBitmap та PatBlt. Розглянемо порядок їх використання в реальному додаткові.

Робота з віртуальним вікном у програмі організується в два етапи: створення віртуального вікна та організація безпосередньої роботи з ним. Створити віртуальне вікно раціонально при обробці повідомлення WM_CREATE, тобто в момент створення вікна додатка. Працювати з цим вікно можна у будь-який час, коли необхідно виконати виведення у вікно.

Фізично віртуальне вікно являє собою растрове зображення, яке зберігається в пам'яті. Робота з цією областю пам'яті організується так само, як і з вікном додатка на екрані монітора. Це означає, що для роботи з ним необхідно створити контекст пристрою пам'яті, сумісний із контекстом вікна. Ця дія реалізується двома функціями: GetDC, за допомогою якої додаток отримує контекст вікна, і CreateCompatibleDC, яка створює сумісний із контекстом вікна контекст пам'яті memdc. Після цього функцією CreateCompatibleBitmap створюється сумісне з реальним вікном на екрані растрове зображення. Його розміри повинні відповідати розміру вікна, для роботи з яким будується растрове зображення. Тому попередньо за допомогою функції GetSystemMetrics повинні бути отримані і передані як параметри у CreateCompatibleBitmap розміри вікна. Функція CreateCompatibleBitmap повертає дескриптор на створене растрове зображення. Після цього функція SelectObject вибирає створене растрове зображення в контекст пам'яті, який, в свою чергу, сумісний із контекстом вікна. Завдяки такому ланцюгу зв'язків звернення до растрового зображення в пам'яті виконується аналогічно зверненню до реального вікна. На практиці це означає, що у всіх функціях, які виводять зображення у вікно, на місці параметра, що відповідає контексту пристрою, необхідно вказувати контекст пристрою пам'яті. Наприклад, функція TextOut буде викликатися так: TextOut(memDC,10,150,@t_o+1,length(t_o));. Видно, що функції TextOut передається не контекст вікна, а контекст віртуального вікна, що і приводить до виведення не в реальне вікно, а у віртуальне, яке є растровим зображенням. Як уже зазначалося, в програмі є лише одне місце, де виконується виведення в реальне вік-

но, – це фрагмент програми, що обробляє повідомлення WM_PAINT. У випадку використання віртуального вікна тут розташовується функція BitBlt, яка копіює вміст растрового зображення з контексту пам'яті в контекст реального вікна. Отже, буде постійно забезпечуватися актуальний вміст вікна додатка. У прикладі 2.2 наведені фрагменти тексту програми, яка демонструє практичну реалізацію способу перемальовування вікна додатка з використанням віртуального вікна.

Приклад 2.2

```
... ... ...
var DC,memDC: hDC;
   Rect: TRect;
   Rgn: HRgn;
   H_Pen:Hpen;
   d_t:string:='Текст виведено функцією DrawText';
   t_o:string:='Текст виведено функцією TextOut';
   p:array[1..4]of TPoint;
   S_DC,maxX,maxY:DWord;
   ps:TPaintStruct;
function WindowProc conv arg_stdcall
(Window:HWND;Mess:UINT;Wp:WParam;Lp:LParam):LRESULT;
begin
 case Mess of
 WM_CREATE:  begin
          maxX:=GetSystemMetrics(SM_CXSCREEN);
          maxY:=GetSystemMetrics(SM_CYSCREEN);
          DC := GetDC(Window);
          memDC:=CreateCompatibleDC(DC);
    SelectObject(memDC,CreateCompatibleBitmap(DC,maxX,maxY));
          PatBlt(memDC,0,0,maxX,maxY,PATCOPY);
          ReleaseDC(Window,DC);    end;
 WM_PAINT:   begin
          DC:= BeginPaint(Window, ps);
          BitBlt(DC,0,0,maxX,maxY,memDC,0,0,SRCCOPY);
          EndPaint(Window, ps);       end;
 WM_DESTROY: begin
```

```
          DeleteDC(memDC);
          PostQuitMessage(0);
          Result := 0;            end;
WM_COMMAND:
   begin
    case LoWord(Wp) of
     IDM_DRAWTEXT:
      begin
      GetClientRect(Window,Rect);
      DrawText(memDC,@d_t+1,length(d_t),Rect,DT_CENTER);
      InvalidateRect(Window,nil,true);   end;
     IDM_TEXTOUT:
      begin
      S_DC:=SaveDC(memDC);
      SelectObject(memDC,GetStockObject(ANSI_VAR_FONT));
      SetTextColor(memDC,RGB(0,0,255));
      TextOut(memDC,10,150,@t_o+1,length(t_o));
      InvalidateRect(Window,nil,true);
      RestoreDC(memDC,S_DC);        end;
     IDM_LINE:
      begin
      MoveToEx(memDC,100,100,nil);
      H_Pen:=CreatePen(PS_SOLID,5,RGB(200,200,200));
      SelectObject(memDC,H_Pen);
      LineTo(memDC,250,140);
      InvalidateRect(Window,nil,true);
      DeleteObject(H_Pen);          end;
     IDM_CURVE:
      begin
      p[1].x:=10;p[2].x:=50;p[3].x:=200;p[4].x:=300;
      p[1].y:=200;p[2].y:=100;p[3].y:=350;p[4].y:=100;
      H_Pen:=CreatePen(PS_SOLID,5,RGB(150,150,150));
      SelectObject(memDC,H_Pen);
      PolyBezier(memDC,p,4);
      InvalidateRect(Window,nil,true);
      DeleteObject(H_Pen);         end;
```

```
        IDM_RECTANGLE:
          begin
          rect.left:=10;rect.right:=90;
          rect.top:=100;rect.bottom:=150;
          FillRect(memDC,rect,CreateSolidBrush(RGB(255,0,0)));
          InvalidateRect(Window,nil,true);       end;
        IDM_ELLIPSE:
          begin
          Rgn:=CreateEllipticRgn(20,20,300,100);
          FillRgn(memDC,Rgn,CreateSolidBrush(RGB(0,255,0)));
          InvalidateRect(Window,nil,true);       end;
        IDM_INFO:
          begin
          MessageBox(Window,'Лабораторна робота з курсу "Операційні системи"', 'Інформаційне повідомлення',MB_OK);
          end;
       end;
     end;
  else Result := DefWindowProc(Window, Mess, Wp, Lp);
  end;
end;
```

... ...

Вікна діалогу у Windows-додатках

Вікна діалогу є важливими та широко використовуваними елементами інтерфейсу користувача, що надає система Windows. Невелика кількість віконних додатків обходиться без використання вікон цього типу. Фізично вікно діалогу являє собою специфічне вікно, робота з яким підтримується на рівні інтерфейсу Win32 API Windows. Основне призначення цього вікна – допомогти користувачу сформувати інформацію, необхідну для керування роботою додатка. Найбільш наочні приклади вікон цього типу – це вікна діалогу, які використовуються в текстовому редакторі. З їх допомогою ви задаєте параметри шрифту або параметри, необхідні для друку документа. Дуже важливо, що розробка таких вікон не вимагає програмування. Для опису вікон діалогу система Windows підтримує спеціальний тип ресурсу.

Для того, щоб створити вікно діалогу, необхідно виконати такі кроки:

- описати вікно діалогу у файлі ресурсів;
- розробити *діалогову* функцію. Дана функція буде обробляти повідомлення, призначені для визначеного у файлі ресурсів вікна діалогу;
- активувати діалог у додатку.

Продемонструємо приклад створення діалогового вікна на основі тексту програми, яка розглядається у даній лабораторній роботі. Забезпечимо роботу підпункту меню „Прямокутник” за допомогою вікна діалогу, задача якого буде полягати у прийнятті від користувача параметрів прямокутної фігури та перемальовуванні її у вікні додатка відповідно до заданих параметрів.

Опис вікна діалогу у файлі ресурсів

Як зазначалося, вікно діалогу являє собою спеціальний об’єкт, призначений для полегшення взаємодії користувача з виконуваною програмою та налаштування її на певні умови функціонування. Вікно діалогу складається з елементарних об’єктів, які називаються *елементами керування*. Система Windows підтримує кілька типів таких об’єктів. У розгляданому прикладі будемо використовувати лише невелику їх частину.

Для опису зовнішнього виду вікна діалогу та забезпечення його інтерфейсу з додатком використовується спеціальний тип ресурсу – DIALOG. На відміну від ресурсу меню, який можна створити вручну, ресурс діалогового вікна краще створювати за допомогою відповідних програмних засобів – редакторів ресурсів. Основна причина тут в тому, що при ручному визначенні розмірів та взаємного розташування елементів керування важко уявити, як це буде виглядати на екрані.

Доповнимо файл ресурсів програми, яка розглядається в даній лабораторній роботі, таким фрагментом:

```
#include "afxres.h"
999 MENU DISCARDABLE
  BEGIN
... ... ... // Опис ресурсів меню
  END
```

```
//Діалог для прямокутника:
1000 DIALOG DISCARDABLE 0,0,186,95
STYLE DS_MODALFRAME|WS_POPUP|WS_CAPTION|WS_SYSMENU
    CAPTION "Прямокутник"
    FONT 8, "MS Sans Serif"
    BEGIN
        DEFPUSHBUTTON "Ok",IDOK,35,72,50,14
        PUSHBUTTON    "Cancel",IDCANCEL,118,72,50,14
       LTEXT "Задайте координати кутів прямокутника:",IDC_STATIC,22,6,174,8
        LTEXT "X",IDC_STATIC,22,39,8,8
        LTEXT "Y",IDC_STATIC,21,55,8,8
        LTEXT "X",IDC_STATIC,104,39,8,8
        LTEXT "Y",IDC_STATIC,104,54,8,8
        EDITTEXT 1001,34,37,20,12,ES_AUTOHSCROLL  //IDC_EDIT1
        EDITTEXT 1002,34,52,20,12,ES_AUTOHSCROLL  //IDC_EDIT2
        EDITTEXT 1003,118,36,20,12,ES_AUTOHSCROLL //IDC_EDIT3
        EDITTEXT 1004,118,52,20,12,ES_AUTOHSCROLL //IDC_EDIT4
        LTEXT "Лівий верхній кут:",IDC_STATIC,10,27,65,8
        LTEXT "Правий нижній кут:",IDC_STATIC,90,27,65,8
    END
```

Після того як файл ресурсів створено, необхідно виконати компіляцію та отримати його двійковий еквівалент. Файл ресурсів, на відміну від попереднього випадку, має особливості. Ці особливості пов’язані з тим, що при описі ресурсу вікна діалогу використовуються символічні імена констант, визначені у файлі afxres.h.

Необхідно також у тексті програми додатка доповнити поле констант, призначивши ідентифікаторам пунктів діалогового вікна відповідні константи із файла ресурсів, які в подальшому будуть передаватися у діалогову процедуру в молодшому слові параметра wParam повідомлення WM_COMMAND:

```
... ... ...
  MY_DIALOG    = 1000;
  IDC_EDIT1    = 1001;
  IDC_EDIT2    = 1002;
  IDC_EDIT3    = 1003;
  IDC_EDIT4    = 1004;
... ... ...
```

Діалогова процедура

У процесі роботи з діалоговим вікном користувач виконує деякі дії, про які за допомогою механізму повідомлень стає відомо додатку. В додатка для кожного діалогового вікна повинна існувати своя процедура, призначена для обробки повідомлень цього вікна. Ця процедура називається *діалоговою процедурою*. Навіть саме примітивне вікно діалогу містить елемент, повідомлення від якого надходить у діалогову процедуру. Зазвичай таке вікно містить кнопку „Ok" або „Cancel". На мові Pascal відповідна діалогова процедура має такий вигляд:

```
function MyDlgProc conv arg_stdcall
(Dialog:HWND;Mess:UINT;Wp:WParam; Lp:LParam):LongInt;
begin
  Result:=0;
  case Mess of
    WM_INITDIALOG:
      Result:=1;
    WM_COMMAND:
       case LoWord(Wp) of
         IDOK: SendMessage(Dialog, WM_CLOSE, 0, 0);
         IDCANCEL: SendMessage(Dialog, WM_CLOSE, 0, 0);
       end;
    WM_CLOSE: EndDialog(Dialog,0);
  end;
end;
```

Слід відзначити, що якщо віконна процедура самостійно обробляє повідомлення, то вона повинна повертати одиничне значення (Result:=1;), якщо ні, то нульове (Result:=0;).

Вікна діалогу, що мають велику кількість елементів керування, повинні, відповідно, мати діалогові процедури, які обробляють повідомлення від цих елементів.

В нашому випадку буде визначене одне вікно діалогу. За допомогою цього вікна користувач може задавати координати кутів прямокутника. На рис. 2.2 наведено вигляд вікна додатка з вікном діалогу для завдання координат прямокутника.

При заданні координат користувач сам повинен контролювати правильність введення даних, оскільки алгоритм програми не передбачає перевірки даних, що вводяться в елементи введення діалогового вікна. Правильні значення, які вводяться в кожне з полів введення, повинні містити всі чотири десяткові цифри, із включенням, при необхідності, старших нулів.

Для обробки повідомлень від цього діалогового вікна програма додатка містить діалогову процедуру MyDlgProc, призначену для введення координат прямокутника. Структура діалогової процедури аналогічна структурі віконної процедури. На початку діалогової процедури знаходиться програмна конструкція, яка визначає тип повідомлення, що надійшло, і в залежності від нього передає керування в певну точку діалогової процедури.

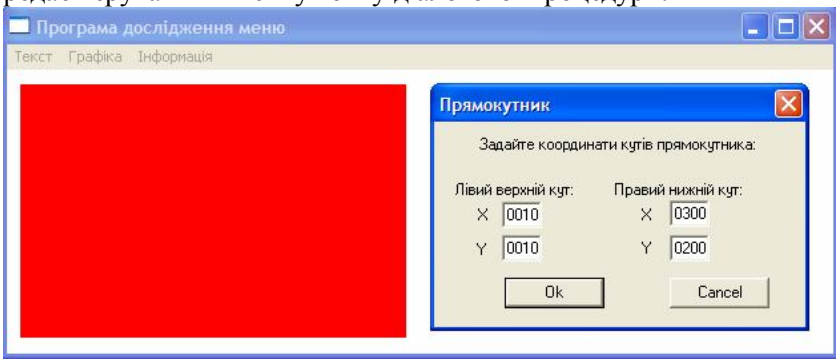


Рис. 2.2. Вікно діалогу для задання координат кутів прямокутника

Головне повідомлення, яке надходить у віконну процедуру – повідомлення WM_COMMAND. Саме воно несе інформацію про дії користувача над елементами керування вікна діалогу. Повідомлення WM_INITDIALOG надходить у діалогову процедуру один раз. Це відбувається в процесі ініціалізації діалогу перед появою вікна діалогу в області вікна додатка. Далі в діалогову процедуру надходить послідовність повідомлень WM_COMMAND. Параметр wParam цього повідомлення містить ідентифікатор того елемента керування, над яким користувач виконав деяку дію, наприклад, клацнув кнопку „Ok” або „Cancel”. Зверніть увагу, що при описі елементів різних вікон діалогу можуть використовува-

тися одні й ті самі ідентифікатори. Більше того, цим ідентифікаторам можуть бути призначені одні й ті самі константи. Саме їх значення передаються у структурі повідомлення для ідентифікації конкретного елемента вікна діалогу. Однакові значення не вносять жодної плутанини у роботу додатка, оскільки для кожного вікна діалогу існує своя процедура.

Діалогова процедура MyDlgProc використовує допоміжну процедуру TextToDigit. Процедура TextToDigit описана у прикладі 2.3 і призначена для перетворення рядка з чотирьох символів десяткових цифр у еквівалентне десяткове число. В даній програмі ця процедура використовується для перетворення символьних рядків, які зчитуються з вікон введення діалогу. Ці символьні рядки логічно являють собою координати прямокутника. Перетворені за допомогою процедури TextToDigit значення цих координат використовуються для роботи функції FillRect.

Активізація діалогу

Для того щоб відобразити діалогове вікно на екрані та виконати з його допомогою деяку роботу, використовується функція DialogBoxParam. Ця функція має такий формат:

```
function DialogBoxParam(
hInstance: HINSTANCE,        // дескриптор додатка
lpTemplateName: LPCTSTR, // вказівник на рядок із заголовком вікна
hWndParent: Hwnd,            // дескриптор вікна
lpDialogFunc: DlgProc,       // вказівник на діалогову процедуру
dwInitParam: Lparam          // значення, яке передається через Lparam
): integer;
```

У фрагменті програми прикладу 2.3 звернення до цієї функції виконується один раз при виборі відповідного пункту меню. За допомогою цієї функції у діалогову процедуру передається 32-бітне значення, яке отримується за допомогою параметра Lparam.

По закінченні роботи вікно діалогу повинно бути закрите функцією EndDialog, яка має формат:

```
EndDialog(
   hDlg: Hwnd,          // дескриптор вікна діалогу
   nResult: integer     // значення, яке повертається
  ): boolean;
```

Зазвичай закриття вікна виконується як реакція на натискання кнопок „Ok” або „Cancel”. У прикладі 2.3 наведено фрагменти тексту додатка, який використовує діалогове вікно для задання координат прямокутника.

Приклад 2.3

```
... ... ...
  MY_DIALOG     = 1000;
  IDC_EDIT1     = 1001;
  IDC_EDIT2     = 1002;
  IDC_EDIT3     = 1003;
  IDC_EDIT4     = 1004;
var DC,memDC: hDC;
    Rect: TRect;
    Rgn: HRgn;
    H_Pen:Hpen;
    d_t:string:=‘Текст виведено функцією DrawText’;
    t_o:string:=‘Текст виведено функцією TextOut’;
    p:array[1..4]of TPoint;
    S_DC,maxX,maxY:DWord;
    ps:TPaintStruct;
    Xstart,Ystart,Xend,Yend:string;
    d,k,i,koord,cod:dword;
procedure TextToDigit(text_:string);
begin
 koord:=0;d:=1000;
 for i:=0 to 3 do begin Val(text_[i],k,Cod);
                 koord:=koord+k*d;d:=round(d/10);
                 end;
end;
function MyDlgProc conv arg_stdcall
(Dialog:HWND;Mess:UINT;Wp:WParam; Lp:LParam):LongInt;
begin
  Result:=0;
  case Mess of
    WM_INITDIALOG:
      Result:=1;
```

```
    WM_COMMAND:
       case LoWord(Wp) of
         IDOK: begin
              GetDlgItemText(Dialog,IDC_EDIT1,@Xstart,5);
              GetDlgItemText(Dialog,IDC_EDIT2,@Ystart,5);
              GetDlgItemText(Dialog,IDC_EDIT3,@Xend,5);
              GetDlgItemText(Dialog,IDC_EDIT4,@Yend,5);
              TextToDigit(Xstart);rect.left:=koord;
              TextToDigit(Xend);rect.right:= koord;
              TextToDigit(Ystart);rect.top:= koord;
              TextToDigit(Yend);rect.bottom:=koord;
              SendMessage(Dialog, WM_CLOSE, 0, 0);
              end;
         IDCANCEL: SendMessage(Dialog, WM_CLOSE, 0, 0);
       end;
    WM_CLOSE: EndDialog(Dialog,0);
  end;
end;
function WindowProc conv arg_stdcall
(Window:HWND;Mess:UINT;Wp:WParam; Lp:LParam):LRESULT;
begin
  case Mess of
  WM_CREATE:
... ... ...
  WM_PAINT:
... ... ...
  WM_DESTROY:
... ... ...
  WM_COMMAND:
      begin
        case LoWord(Wp) of
          IDM_DRAWTEXT:
... ... ...
          IDM_TEXTOUT:
... ... ...
```

```
        IDM_LINE:
... ... ...
        IDM_CURVE:
... ... ...
        IDM_RECTANGLE:
          begin
          DialogBoxParam(hInstance,MAKEINTRESOURCE(1000),
                                              Window,@MyDlgProc,Wp);
          FillRect(memDC,rect,CreateSolidBrush(RGB(255,0,0)));
          InvalidateRect(Window,nil,true);
          end;
        IDM_ELLIPSE:
... ... ...
        IDM_INFO:
... ... ...
```

Практична частина

1. Складіть програми, наведені у прикладах 2.1, 2.2 та 2.3 теоретичної частини даної лабораторної роботи. Запустіть програми та переконайтеся, що вони працюють правильно (згідно з описом алгоритму їх роботи).

2. Оформіть звіт із виконаної лабораторної роботи, який повинен містити текст програм, та дайте відповіді на контрольні запитання.

Контрольні запитання та завдання

1. Що являють собою ресурси Windows-додатків?
2. Яким чином можна створити ресурси Windows-додатків?
3. Опишіть процес організації меню у Windows-додатках.
4. Як організувати перемальовування зображення вікна Windows-додатка?
5. Що являють собою вікна діалогу?
6. Опишіть особливості побудови та функціонування діалогової процедури.
7. Як активувати діалогове вікно?

Лабораторна робота № 3
Дослідження особливостей використання API-функцій для роботи із процесами та потоками у Windows

Теоретична частина

Поняття процесу і потоку

На сьогоднішній день загальноприйнятий погляд на операційну систему (ОС) як на систему, що забезпечує паралельне (або псевдопаралельне) виконання набору послідовних процесів або просто процесів. Завдання ОС полягає в тому, щоб організувати їх підтримку, яка передбачає, що кожен процес одержить усі необхідні йому ресурси (місце в пам'яті, процесорний час і т.д.). Вважається також, що незалежні процеси не повинні впливати один на одного, а процеси, яким необхідно обмінюватися інформацією, зможуть зробити це шляхом міжпроцесної взаємодії.

З лекційного курсу теорії операційних систем відомо, що процес є динамічним об'єктом, що описує виконання програми. Процесу виділяються системні ресурси: закритий адресний простір, семафори, комунікаційні порти, файли і т.д. Процес характеризується поточним станом (виконання, очікування, готовність і т.д.).

Для опису такого складного динамічного об'єкта ОС підтримує набір структур, головну з яких прийнято називати *блоком управління процесом* (PCB, Process control block). До складу PCB зазвичай включають:

- стан, в якому знаходиться процес;
- програмний лічильник процесу або, іншими словами, адреса команди, яка повинна бути виконана для нього наступною;
- вміст регістрів процесора;
- дані, необхідні для планування використання процесора і управління пам'яттю (пріоритет процесу, розмір і розташування адресного простору і т. д.);
- облікові дані (ідентифікаційний номер процесу, з яким користувач ініціював його роботу, загальний час використання процесора даним процесом і т. д.);

- інформацію про пристрої введення-виведення, пов'язані з процесом (наприклад, які пристрої закріплені за процесом; таблиця відкритих файлів).

Блок управління процесом є моделлю процесу для операційної системи. Будь-яка операція, що виконується операційною системою над процесом, викликає певні зміни в РСВ. Псевдопаралельне виконання процесів передбачає періодичне припинення поточного процесу і його подальше відновлення. Для цього потрібно вміти зберігати частину даних із РСВ, які зазвичай називають *контекстом* процесу, а операцію по збереженню даних одного процесу і відновленню даних іншого називають перемиканням контекстів. Перемикання контексту не має відношення до корисної роботи, що виконується процесами, і час, витрачений на це, скорочує корисний час роботи процесора.

Потоки

Класичний процес містить у своєму адресному просторі одну програму. Проте в багатьох ситуаціях доцільно підтримувати в єдиному адресному просторі процесу кілька виконавчих програм (потоків команд або просто потоків), що працюють із загальними даними і ресурсами. В цьому випадку процес можна розглядати як контейнер ресурсів, а всі проблеми, пов'язані з динамікою виконання, розв'язуються на рівні потоків. Зазвичай кожен процес починається з одного потоку, а інші (при необхідності) створюються в ході виконання. Тепер уже не процес, а потік характеризується станом, потік є одиницею планування, процесор перемикається між потоками, і необхідно зберігати контекст потоку (що значно простіше, ніж збереження контексту процесу).

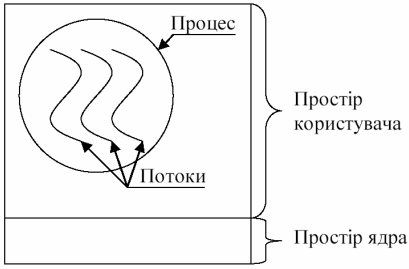


Рис. 3.1. Процес із кількома потоками

Подібно до процесів, потоки (нитки, threads) у системі описуються структурою даних, яку зазвичай називають *блоком управління потоком* (thread control block, TCB).

Реалізація процесів

Внутрішня будова процесів в ОС Windows

У 32-розрядній версії системи у кожного процесу є 4-гігабайтний адресний простір, в якому код користувача займає нижні 2 гігабайти (у серверах – 3 Гбайти). У своєму адресному просторі, який є набором регіонів і описується спеціальними структурами даних, процес містить потоки, облікову інформацію і посилання на ресурси, які спільно використовуються всіма потоками процесу.

Блок управління процесом (PCB) реалізований у вигляді набору зв'язаних структур, головна з яких називається блоком процесу EPROCESS. Відповідно, кожен потік також представлений набором структур із блоком потоку ETHREAD. Ці набори даних, за винятком блоків змінних оточення процесу і потоку (PEB і TEB), існують у системному адресному просторі. Спрощена схема структур даних процесу показана на рис. 3.2.

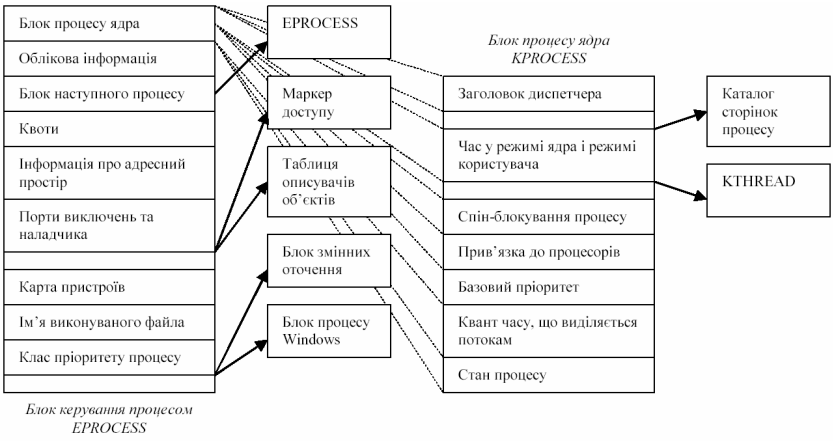


Рис. 3.2. Керуючі структури даних процесу

Вміст блоку EPROCESS детально описаний у відповідній літературі. Блок KPROCESS (на рис. справа), блок змінних оточення

процесу (PEB) і структура даних, підтримувана підсистемою Win32 (блок процесу Win32), містять додаткові відомості про об'єкт "процес".

Ідентифікатор процесу кратний чотирьом і використовується в ролі байтового індексу в таблицях ядра на одному рівні з іншими об'єктами.

Створення процесу

Зазвичай процес створюється іншим процесом викликом Win32-функції CreateProcess (а також CreateProcessAsUser і CreateProcessWithLogonW). Створення процесу здійснюється в кілька етапів.

На першому етапі, що виконується бібліотекою *kernel32.dll* у режимі користувача, на диску відшукується потрібний файл-образ, після чого створюється об'єкт "розділ" пам'яті для його проектування на адресний простір нового процесу.

На другому етапі виконується звернення до системного сервісу *NtCreateProcess* для створення об'єкта "процес". Формуються блоки EPROCESS, KPROCESS і блок змінних оточення PEB. Менеджер процесів ініціалізував у блоці процесу маркер доступу (копіюючи аналогічний маркер батьківського процесу), ідентифікатор та інші поля.

На третьому етапі в об'єкті, що вже повністю проініціалізував, "процес" необхідно створити первинний потік. Це, за допомогою системного сервісу *NtCreateThread*, робить бібліотека *kernel32.dll*.

Потім *kernel32.dll* посилає підсистемі Win32 повідомлення, яке містить інформацію, необхідну для виконання нового процесу. Дані про процес і потік поміщаються, відповідно, в список процесів і список потоків даного процесу, потім встановлюється пріоритет процесу, створюється структура, що використовується тією частиною підсистеми Win32, яка працює в режимі ядра, і т.д.

Нарешті, запускається первинний потік, для чого формуються його початковий контекст і стек, і виконується запуск стартової процедури потоку режиму ядра *KiThreadStartup*. Після цього стартовий код програми передає управління головній функції програми, що запускається.

Функція CreateProcess

Отже, якщо додаток має намір створити новий процес, один із його потоків повинен звернутися до Win32-функції CreateProcess.

```
BOOL CreateProcess(
  PCTSTR pszApplicationName
  PTSTR pszCommandLine
  PSECURITY_ATTRIBUTES psaProcess
  PSECURITY_ATTRIBUTES psaThread
  BOOL bInheritHandles
  DWORD fdwCreate
  PVOID pvEnvironment
  PCTSTR pszCurDir
  PSTARTUPINFO psiStartInfo
  PPROCESS_INFORMATION ppiProcInfo);
```

Опис параметрів функції можна знайти у відповідній довідниковій літературі.

Формально ОС Windows не підтримує жодної ієрархії процесів, наприклад, відношень "батьківський-дочірній". Проте негласна ієрархія, що полягає в тому, хто чиїм дескриптором (описувачем) володіє, все ж таки існує. Наприклад, володіння дескриптором процесу дозволяє впливати на його адресний простір і функціонування. В даному випадку описувач дочірнього процесу повертається утворюючому процесу у складі параметра ppiPricInfo. Хоча він не може бути безпосередньо переданий іншому процесу, але існує можливість передати іншому процесу його дублікат. Таким шляхом при необхідності в групі процесів може бути сформована потрібна ієрархія.

Завершення процесу може бути здійснено різними способами, наприклад за допомогою функцій ExitProcess, TerminateProcess. Однак єдиним способом, що гарантує коректне очищення всіх ресурсів, є повернення управління вхідною функцією первинного потоку. Крім перерахованих, система виконує багато корисних функцій, що реалізовують API для управління процесами. Їх повний перелік міститься у відповідній довідниковій літературі.

При завершенні процесу зіставлений із ним об’єкт ядра "процес" не звільняється доти, поки не будуть закриті всі зовнішні посилання на цей об’єкт.

Дослідження програми створення та завершення процесів

У прикладі 3.1 наведено текст програми, яка демонструє можливість створення та завершення процесів. При виборі елемента меню „Новий процес” створюється процес, в якому запускається звичайний текстовий редактор Notepad (можна запустити будь-яку іншу програму, наприклад калькулятор – calc). Максимум може бути запущено 10 процесів. По команді „Завершити процес” процеси завершуються в порядку, зворотному до їх створення. Результати створення процесу, що беруться зі структури типу PROCESS_INFORMATION, відображаються у вікні повідомлень. Тут, зокрема, виводиться номер ідентифікатора процесу, який також можна побачити, запустивши програму для перегляду задач, які виконуються в даний момент часу на комп’ютері («Пуск» → «Программы» → «Стандартные» → «Служебные» → «Сведения о системе» → «Программная среда» → «Выполняемые задачи»).

Приклад 3.1

```
uses Windows,Messages;
{$r menu.res}
const
  MY_MENU           = 999;
  IDM_New_Process    = 100;
  IDM_Kill_Process   = 101;
  IDM_Exit          = 102;
  max=10;
var St_Info:TStartupInfo;
    i,ProcNumb:integer:=0;
    Proc_Info:array[0..max]of TProcessInformation;
    My_Mess:array[1..200]of char;
    help_str1,help_str2:string;
    Sub_Menu:hMenu;
function WindowProc conv arg_stdcall
(Window:HWND;Mess:UINT;Wp:WParam;Lp:LParam):LRESULT;
```

```
begin
  case Mess of
      WM_CREATE:Sub_Menu:=GetSubMenu(GetMenu(Window),0);
      WM_COMMAND:
      begin
       case LoWord(Wp) of
           IDM_NEW_PROCESS:
           if ProcNumb<max then begin
             St_Info.cb:=sizeof(St_Info);
             St_Info.lpReserved:=nil;
             St_Info.lpDesktop:=nil;
             St_Info.lpTitle:=nil;
             St_Info.dwFlags:=STARTF_USESHOWWINDOW;
             St_Info.wShowWindow:=SW_SHOWNORMAL;
             St_Info.cbReserved2:=0;
             St_Info.lpReserved2:=nil;
if CreateProcess(nil,'Notepad.exe',nil,nil,false,0,nil,nil,
St_Info,Proc_Info[ProcNumb]) then begin
inc(ProcNumb);help_str1:='';
Str(Proc_Info[ProcNumb-1].hProcess,help_str2);
help_str1:=help_str1+'Дескриптор процесу: '+help_str2;
Str(Proc_Info[ProcNumb-1].hThread,help_str2);
help_str1:=help_str1+', Дескриптор потоку: '+help_str2;
Str(Proc_Info[ProcNumb-1].dwProcessId,help_str2);
help_str1:=help_str1+', Ідентифікатор процесу: '+help_str2;
Str(Proc_Info[ProcNumb-1].dwThreadId,help_str2);
help_str1:=help_str1+', Ідентифікатор потоку: '+help_str2;
for i:=1 to Length(My_Mess) do My_Mess[i]:=help_str1[i];
MessageBox(Window,@My_Mess,'Новий процес створено!',
MB_OK);
EnableMenuItem(Sub_Menu,IDM_Kill_Process,MF_BYCOMMAND
+ MF_ENABLED);
end else MessageBox(Window,'Неможливо створити процес',
'Створення процесу',MB_OK);
end else MessageBox(Window,'Дуже багато створено процесів',
'Створення процесу',MB_OK);
```

```
          IDM_Kill_Process: if ProcNumb>0 then
            if TerminateProcess(Proc_Info[ProcNumb-1].hProcess,0)
then begin
              Dec(ProcNumb);
              if ProcNumb=0 then
EnableMenuItem(Sub_Menu,IDM_Kill_Process,
MF_BYCOMMAND + MF_GRAYED)
              end else MessageBox(Window,'Неможливо завершити
процес','Завершення процесу',MB_OK)
                else MessageBox(Window,'Немає більше процесів',
'Завершення процесу',MB_OK);
          IDM_Exit: SendMessage(Window,WM_CLOSE,0,0);
       end;
      end;
      WM_DESTROY: PostQuitMessage(0);
  else Result := DefWindowProc(Window, Mess, Wp, Lp);
  end;
end;
var wc: TWndClass;wnd:HWnd;Msg: TMsg;Menu: hMenu;
begin
  FillChar(wc, SizeOf(wc), 0);
  with wc do begin
    style:=CS_HREDRAW + CS_VREDRAW;
    lpfnWndProc := @WindowProc;
    cbClsExtra := 0;
    cbWndExtra := 0;
    hInstance := System.hInstance;
    hIcon := LoadIcon(THandle(NIL), IDI_APPLICATION);
    hCursor := LoadCursor(THandle(NIL), IDC_ARROW);
    hbrBackGround := GetStockObject(WHITE_BRUSH);
    lpszMenuName := MAKEINTRESOURCE(MY_MENU);
    lpszClassName := 'Лабораторна робота № 3';
          end;
  if RegisterClass(wc) = 0 then Exit;
wnd := CreateWindow(wc.lpszClassName,'Програма дослідження
створення та завершення процесів',
```

```
WS_OVERLAPPEDWINDOW,CW_USEDEFAULT,
CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,0,0,
HInstance,nil);
  ShowWindow(wnd, SW_RESTORE);
  UpdateWindow(wnd);
  while GetMessage(Msg,0,0,0) do
  begin
    TranslateMessage(Msg);
    DispatchMessage(Msg);
  end;
end.
```

У цій програмі використовується файл ресурсів (menu.rc):

```
999 MENU DISCARDABLE
BEGIN
POPUP "&Процеси"
BEGIN
MENUITEM "&Новий процес", 100 //IDM_New_Process
MENUITEM "&Завершити процес",101,GRAYED //IDM_Kill_Process
MENUITEM SEPARATOR
MENUITEM "&Вихід",  102 //IDM_Exit
END
END
```

Реалізація потоків

Стани потоків

Кожен новий процес містить принаймні один потік, решта потоків створюються динамічно. Потоки складають основу планування і можуть: виконуватися на одному з процесорів, чекати події або знаходитися в якомусь іншому стані.

Зазвичай у стані "Готовності" є черги готових до виконання (running) потоків. У даному випадку цей стан розпадається на три складових. Це, власне, стан "Готовності (Ready)"; стан "Готовий. Відкладений (Deferred Ready)", що означає, що потік вибраний для виконання на конкретному процесорі, але поки не запланований до виконання; і, нарешті, стан "Простоює (Standby)", в якому може знаходитися тільки один вибраний до виконання потік для кожного процесора в системі.

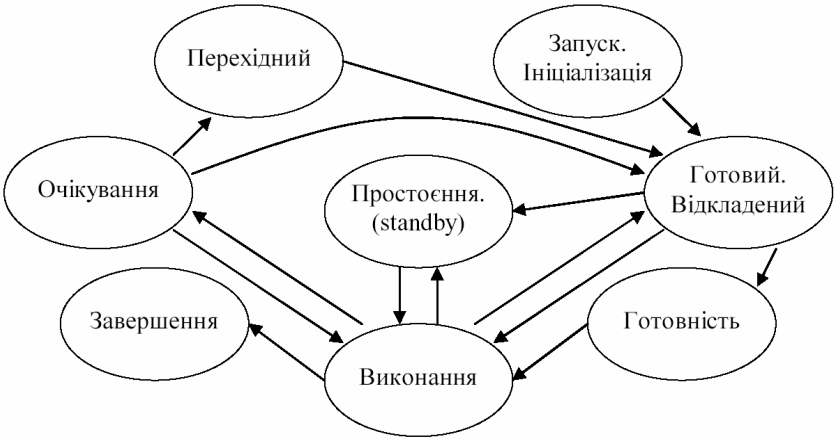


Рис. 3.3. Стани потоків в ОС Windows

У стані "Очікування (Waiting)" потік блокований і чекає якої-небудь події, наприклад завершення операції введення-виведення. При настанні цієї події потік переходить у стан "Готовності". Цей шлях може проходити через проміжний "Перехідний (Transition)" стан в тому випадку, якщо стек ядра потоку вивантажений із пам'яті.

Код ядра виконується в контексті поточного потоку. Це означає, що при перериванні, системному виклику і т д., тобто коли процесор переходить у режим ядра і управління передається ОС, перемикання на інший потік (наприклад, системний) не відбувається. Контекст потоку при цьому зберігається, оскільки операційна система все ж таки може ухвалити рішення про зміну характеру діяльності і перемиканні на інший потік. Внаслідок цього в деякій літературі по операційним системам стан "Виконання" розділяють на "Виконання в режимі користувача" і "Виконання в режимі ядра".

Окремі характеристики потоків

Ідентифікатори потоків, так само як і ідентифікатори процесів, кратні чотирьом, вибираються з того ж простору, що й ідентифікатори процесів, і з ними не перетинаються.

Як уже зазначалося, коли потік звертається до системного виклику, то перемикається в режим ядра, після чого продовжує ви-

конуватися той же потік, але вже в режимі ядра. Тому в кожного потоку два стеки, один працює в режимі ядра, інший – у режимі користувача. Один і той самий стек не може використовуватися і в режимі користувача, і в режимі ядра. Будь-який потік може робити все що завгодно зі своїм власним стеком (стеком режиму користувача), зокрема організовувати декілька стеків і перемикатися між ними. Потік сам може визначати розмір свого стека. При цьому не можна гарантувати, що стек матиме достатній розмір, щоб код ядра виконався без жодних проблем. Оскільки виникнення виняткової ситуації в режимі ядра може призвести до краху всієї системи, необхідно виключити таку можливість, що і здійснюється шляхом організації окремого стека для режиму ядра. Оскільки в режимі ядра можуть одночасно знаходитися декілька потоків і між ними може відбуватися перемикання, в кожного з них повинен бути окремий стек режиму ядра.

Крім стану, ідентифікатора і двох стеків, у кожного потоку є контекст, маркер доступу, а також невелика власна пам'ять для зберігання локальних змінних, наприклад для запам'ятовування коду помилки. Оскільки процес є контейнером ресурсів усіх вхідних у нього потоків, будь-який потік може дістати доступ до всіх об'єктів свого процесу, незалежно від того, яким потоком даного процесу цей об'єкт створений.

Волокна і завдання

Перемикання між потоками займає досить багато часу, тому для полегшеного псевдопаралелізму в системі підтримуються волокна (fibers). Наявність волокон дозволяє реалізувати власний механізм планування, не використовуючи вбудований механізм планування потоків на основі пріоритетів. ОС не знає про зміну волокон, для управління волокнами немає і справжніх системних викликів, проте є виклики Win32 API ConvertThreadToFiber, CreateFiber, SwitchToFiber і т.д. Докладніше функції, пов'язані з волокнами, описані у відповідній документації.

У системі є також завдання (job object), які забезпечують управління одним або кількома процесами як групою.

Внутрішня будова потоків

Перейдемо до формального опису потоків. Матеріал цього розділу однаковою мірою стосується як звичайних потоків режиму користувача, так і системних потоків режиму ядра.

Подібно до процесів, кожен потік має свій блок управління, реалізований у вигляді набору структур, головна з яких – ETHREAD – показана на рис. 3.4.

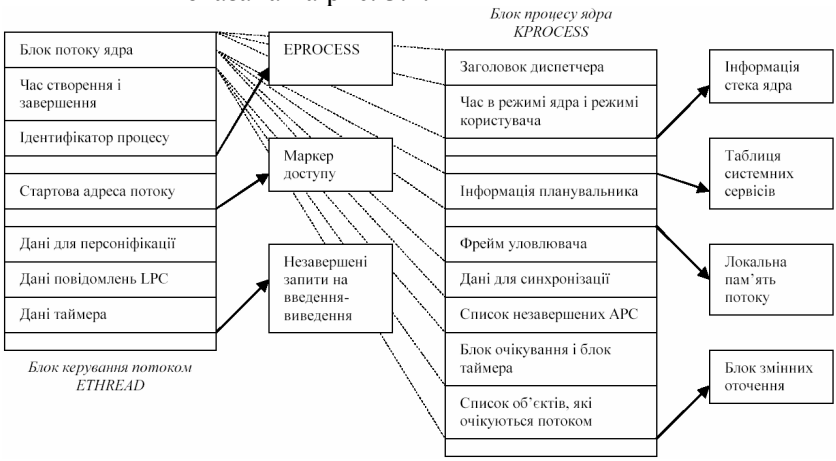


Рис. 3.4. Керуючі структури даних потоку

Зображені на рис. 3.4 структури, за винятком блоків змінних оточення потоку (TEB), існують у системному адресному просторі. Крім цього, паралельна структура для кожного потоку, створеного у Win32-процесі, підтримується процесом Csrss підсистеми Win32. У свою чергу, частина підсистеми Win32, що працює в режимі ядра (Win32k.sys), підтримує для кожного потоку структуру W32THREAD.

Блок потоку ядра KTHREAD містить інформацію, необхідну ядру для планування потоків і їх синхронізації з іншими потоками. Перегляд структур даних потоку може бути здійснений налагодником. Більш детально даний матеріал викладений у відповідній літературі.

Створення потоків

Створення потоку ініціюється Win32-функцією CreateThread, яка знаходиться в бібліотеці Kernel32.dll. При цьому створюється об'єкт ядра "потік", що зберігає статистичну інформацію про створюваний потік. У адресному просторі процесу виділяється пам'ять під стек потоку користувача. Потім ініціюється апаратний контекст потоку (нижче є опис відповідної структури CONTEXT).

Услід за цим створюється блок управління потоком разом із супутніми структурами, формується стек ядра потоку, і про створення потоку повідомляється підсистема Win32. Нарешті, потоку, який викликається, повертається описувач створюваного потоку і передається управління, а новому потокові може бути виділено процесорний час.

Функція CreateThread

Таким чином, якщо первинний потік процесу створюється при виклику функції CreateProcess, то для створення додаткових потоків потрібно викликати функцію CreateThread:

```
HANDLE CreateThread (
PSECURITY_ATTRIBUTES psa
DWORD cbStack
PTHREAD_START_ROUTINE  pfnStartAddr
PVOID pvParam
DWORD fdwCreate
PDWORD pdwThreadID);
```

Дослідження прикладу програми створення потоку

Програма, текс якої наведений нижче, створює новий потік і передає йому параметр, числове значення якого цей потік виводить на екран (**у консольному режимі**).

Приклад 3.2

```
Uses windows;
procedure MyThread(lpParam:^DWORD);
begin
 writeln('Parameter = ',lpParam^);
end;
var ThreadId:DWORD;
```

```
   ThreadParameter:DWORD=10;
   hThread:HANDLE;
begin
 hThread:=CreateThread(
   NIL,                  // атрибути безпеки за замовчуванням
   0,                     // розмір стека за замовчуванням
   @MyThread  ,    // вказівник на процедуру створюваного потоку
   @ThreadParameter,  // аргумент, що передається функції потоку
   0,                     // прапорці створення за замовчуванням
   ThreadId);             // отриманий ідентифікатор потоку
   if (@hThread = NIL)then writeln('CreateThread failed.');
   readln;
   CloseHandle(hThread);
end.
```

Як самостійну вправу рекомендується написати програму, яка ілюструє простоту організації міжпотокового обміну в рамках одного процесу, наприклад, обмін через набір загальних глобальних даних.

Завершення потоку можна організувати різними способами, зокрема, за допомогою функцій ExitThread або TerminateThread. Рекомендований спосіб – повернення управління функцією потоку. Це єдиний спосіб, який гарантує коректне очищення всіх ресурсів, що належали потоку.

Подібно до процесів при завершенні потоку зіставлений із ним об'єкт ядра "потік" не звільняється доти, поки не будуть закриті всі зовнішні посилання на цей об'єкт.

Контекст потоку, перемикання контекстів

Особливу роль у структурах даних, що описують потоки, відіграє контекст потоку. Інформацію, що входить до складу контексту, необхідно періодично зберігати і відновлювати в разі виникнення різних подій, наприклад при перемиканні потоків. Зазвичай збереженню і подальшому відновленню підлягають:

- програмний лічильник, регістр стану і вміст решти регістрів процесора;
- покажчики на стек ядра і стек користувача;

- покажчики на адресний простір, в якому виконується потік (каталог таблиць сторінок процесу).

Ця інформація зберігається в поточному стекові ядра потоку.

Контекст відображає стан регістрів процесора на момент останнього виконання потоку і зберігається в структурі CONTEXT, визначеній у заголовному файлі WinNT.h. Елементи цієї структури відповідають регістрам процесора, наприклад, для процесорів x86 до її складу входять Eax, Ebx, Ecx, Edx і т д.. Win32-функція GetThreadContext дозволяє отримати поточний стан контексту, а функція SetThreadContext – задати новий вміст контексту. Перед цією операцією потік рекомендується припинити.

Крім перерахованих, у системі є багато корисних функцій, що реалізовують API для управління потоками. Їх повний перелік міститься у відповідній літературі.

Дослідження прикладу створення багатопотокового додатка

Наведений у прикладі 3.3. Windows-додаток демонструє створення і паралельну роботу трьох потоків.

Приклад 3.3

```
uses Windows,Messages;
const UM_THREAD_DONE=WM_USER;
type ThreadManager =  packed record
    hwndParent: HWND;
    name: string;
    nValue:integer;      end;
var DC: hDC;
   ps:TPaintStruct;
   hThreadA,hThreadB,hThreadC:HANDLE;
   tex:array[1..3]of string;
   Tm:array[1..3]of ThreadManager;
   y:array[1..3]of integer;
   i:integer;
procedure Thread_Sub_Func(count:dword);
begin
Tm[i].name:=tex[i];Tm[i].nValue:=count;
```

```
SendMessage(Tm[i].hwndParent,UM_THREAD_DONE,
                                                wParam(@Tm[i]),0);
end;
procedure ThreadFuncA;
var count:dword=0;
begin count:=0;for i:=1 to 100000000 do
inc(count);i:=1;Thread_Sub_Func(count);end;
procedure ThreadFuncB;
var count:dword=0;
begin count:=0;for i:=1 to 400000000 do
inc(count);i:=2;Thread_Sub_Func(count);end;
procedure ThreadFuncC;
var count:dword=0;
begin count:=0;for i:=1 to 800000000 do
inc(count);i:=3;Thread_Sub_Func(count);end;
function WindowProc conv
arg_stdcall(Window:HWND;Mess:UINT;Wp:WParam;Lp:LParam):L
RESULT;
begin
  case Mess of
      WM_CREATE:
      begin
      Tm[1].hwndParent:=Window;
hThreadA:=CreateThread(NIL,0,@ThreadFuncA,
                                                @Tm[1],0,hInstance);
      if (@hThreadA = NIL)then MessageBox(Window,'Помилка
створення потоку A', nil,MB_OK);
      Tm[2].hwndParent:=Window;
hThreadB:=CreateThread(NIL,0,@ThreadFuncB,
                                                @Tm[2],0,hInstance);
      if (@hThreadB = NIL)then MessageBox(Window,'Помилка
створення потоку B', nil,MB_OK);
      Tm[3].hwndParent:=Window;
      hThreadC:=CreateThread(NIL,0,@ThreadFuncC,
                                                @Tm[3],0,hInstance);
```

```
      if (@hThreadC = NIL)then MessageBox(Window,'Помилка
створення потоку C', nil,MB_OK);       end;
      UM_THREAD_DONE:
      begin
      Str(Tm[i].nValue,tex[i]);
      tex[i]:=Tm[i].name+': Counter = '+tex[i];
      InvalidateRect(Window,nil,true);       end;
      WM_PAINT:
      begin
      DC:= BeginPaint(Window, ps);
      for i:=1 to 3 do
      TextOut(DC,20,y[i],@tex[i]+1,length(tex[i]));
      EndPaint(Window, ps);       end;
      WM_DESTROY:
      begin
      CloseHandle(hThreadA);
      CloseHandle(hThreadB);
      CloseHandle(hThreadC);
      PostQuitMessage(0);       end;
  else Result := DefWindowProc(Window, Mess, Wp, Lp);
  end; end;
var wc: TWndClass;wnd:HWnd;Msg: TMsg;Menu: hMenu;
begin y[1]:=30;y[2]:=60;y[3]:=90;
tex[1]:='Потік A';tex[2]:='Потік B';tex[3]:='Потік C';
  FillChar(wc, SizeOf(wc), 0);
  with wc do begin
    style:=CS_HREDRAW + CS_VREDRAW;
    lpfnWndProc := @WindowProc;
    cbClsExtra := 0;
    cbWndExtra := 0;
    hInstance := System.hInstance;
    hIcon := LoadIcon(THandle(NIL), IDI_APPLICATION);
    hCursor := LoadCursor(THandle(NIL), IDC_ARROW);
    hbrBackGround := GetStockObject(WHITE_BRUSH);
    lpszMenuName := nil;
    lpszClassName := 'Лабораторна робота № 3';     end;
```

```
  if RegisterClass(wc) = 0 then Exit;
    wnd := CreateWindow(wc.lpszClassName,'Програма дослідження
багатопотокового додатка', WS_OVERLAPPEDWINDOW,
200,200,450,180,0,0,HInstance,nil);
  ShowWindow(wnd, SW_RESTORE);
  UpdateWindow(wnd);
  while GetMessage(Msg,0,0,0) do  begin
    TranslateMessage(Msg);
    DispatchMessage(Msg);
  end; end.
```

У додатку визначена структура ThreadManager, об'єкти якої Tm[1], Tm[2], Tm[3] використовуються для взаємозв'язку дочірніх потоків із первинним потоком. Адреси цих об'єктів передаються як четвертий параметр при викликах функції CreateThread.

У блоці обробки повідомлення WM_CREATE створюються три дочірніх потоки з дескрипторами hThreadA, hThreadB, hThreadC.

Функції потоків ThreadFuncA, ThreadFuncB, ThreadFuncC працюють за одним і тим самим сценарієм. У кожній із них є локальний лічильник count, який інкрементується в циклі. Різниця полягає лише у кількості повторень циклу: 100000000 400000000 800000000 разів відповідно. Після завершення циклу кожна функція посилає вікну первинного потоку повідомлення користувача UM_THREAD_DONE. Отримавши повідомлення UM_THREAD_DONE, віконна функція WindowProc виводить у своє вікно інформацію про потік, який завершився. На рис. 3.5 показано результат запуску додатка.

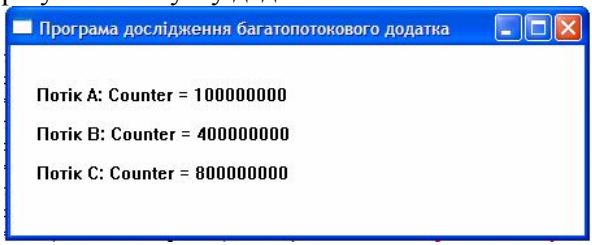


Рис. 3.5. Результати виконання програми

Хоча потоки запущено майже одночасно (порядок запуску: потік C, потік B, потік A), швидше за всіх завершиться потік A, потім – потік B і останнім – потік C. У цій послідовності інформація про них виводиться в головне вікно додатка.

Практична частина

1. Складіть програми, наведені у прикладах 3.1, 3.2 та 3.3 теоретичної частини даної лабораторної роботи. Запустіть програми та переконайтеся, що вони працюють правильно (згідно з описом алгоритму їх роботи).

2. Оформіть звіт із виконаної лабораторної роботи, який повинен містити текст програм, та дайте відповіді на контрольні запитання.

Контрольні запитання та завдання

1. Що собою являють процеси та потоки?
2. Що собою являє блок управління процесом та що входить до його складу?
3. Що собою являє блок управління потоком та що входить до його складу?
4. Опишіть особливості реалізації процесів у операційній системі Windows.
5. Опишіть особливості реалізації потоків у операційній системі Windows.
6. Що собою являють волокна (легковагові потоки) і яке їх призначення?
7. Опишіть основні API-функції керування процесами та потоками у Windows.
8. Опишіть стани потоків в ОС Windows.
9. Що відбувається під час переходу від виконання одного потоку до іншого?

Лабораторна робота № 4
Дослідження особливостей планування потоків у операційній системі Windows

Теоретична частина

Вибір поточного потоку з кількох активних потоків, що намагаються отримати доступ до процесора, називається *плануванням*. Планування – дуже важлива і критична для продуктивності операція, тому система надає багато інструментів для її гнучкої настройки.

Вибраний для виконання потік працює протягом якогось періоду, званого *квантом*, після закінчення якого потік витісняється, тобто процесор передається іншому потоку. Передбачається, що потік не знає, в який момент він буде витіснений. Потік також може бути витіснений, навіть якщо його квант ще не закінчився. Це відбувається, коли до виконання готовий потік із вищим пріоритетом. Процедура планування зазвичай пов'язана з досить витратною процедурою диспетчеризації – перемиканням процесора на новий потік, тому планувальник повинен піклуватися про ефективне використання процесора. Приналежність потоків до процесу при плануванні не враховується, тобто одиницею планування в ОС Windows є саме потік. Запуск процедури планування зручно проілюструвати на спрощеній (у порівнянні з діаграмою, зображеною на рис. 3.3 лабораторної роботи № 3) діаграмі станів потоку (див. рис. 4.1).

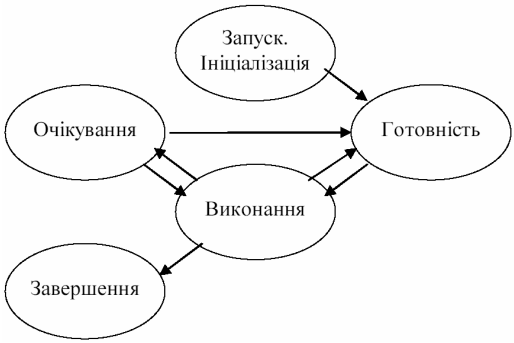


Рис. 4.1. Спрощена діаграма станів потоків в ОС Windows

Найбільш важливим питанням планування є вибір моменту для ухвалення рішення. У ОС Windows запуск процедури планування викликається однією з наступних подій.

Це, *по-перше*, події, пов'язані зі звільненням процесора.

(1) Завершення потоку

(2) Перехід потоку в стан готовності у зв'язку з тим, що його квант часу закінчився

(3) Перехід потоку в стан очікування

По-друге, це події, в результаті яких поповнюється або може поповнитися черга потоків у стані готовності.

(4) Потік вийшов зі стану очікування

(5) Потік щойно створений

(6) Діяльність поточного потоку може мати наслідком виведення іншого потоку зі стану очікування.

У останньому випадку виведений зі стану очікування потік може відразу ж почати виконуватися, якщо має високий пріоритет.

Нарешті, процедура планування може бути запущена, якщо змінюється пріоритет потоку в результаті виклику системного сервісу або самої Windows, а також якщо змінюється прив'язка (affinity) потоку до процесора, через що потік не може більше виконуватися на поточному процесорі.

Зауважимо, що перемикання з режиму користувача в режим ядра (і назад) не впливає на планування потоку, оскільки контекст у цьому випадку не змінюється.

В результаті операції планування система може визначити, який потік виконувати наступним, і перемкнути контексти старого і нового потоків. У системі немає центрального потоку планувальника. Програмний код, що відповідає за планування і диспетчеризацію, розосереджений по ядру. У випадках 1-3 процедури планування працюють у контексті поточного потоку, який запускає програму планувальника для вибору наступника і потенційного завантаження його контексту.

Переведення потоку зі стану очікування в стан готовності (варіант 4) може бути наслідком переривання, що свідчить про закінчення операції введення-виведення. В цьому випадку процеду-

ра планування може бути відкладена (deffered procedure call) до закінчення виконання високопріоритетного системного коду.

Іноді подібний перехід відбувається в результаті діяльності іншого потоку, який, наприклад, виконав операцію up на семафорі (приклад 6-го варіанта). Хоча цей інший потік і може продовжити роботу, він повинен запустити процедуру планування, оскільки в черзі готовності можуть опинитися потоки з вищим пріоритетом. Із тих самих причин планування здійснюється в разі запуску нового потоку.

Алгоритми планування

Пріоритети

У ОС Windows реалізовано витісняюче пріоритетне планування, коли кожному потоку привласнюється певне числове значення – пріоритет, відповідно до якого йому виділяється процесор. Потоки з однаковими пріоритетами плануються згідно з алгоритмом Round Robin (карусель). Важливою перевагою системи є можливість витіснення потоків, що працюють у режимі ядра – код виконавчої системи повністю реєнтерабельний. Не витісняються лише потоки, що утримують спін-блокування (див. лабораторну роботу № 6, присвячену синхронізації потоків). Тому спін-блокування використовуються з великою обережністю та встановлюються на мінімальний час.

У системі передбачено 32 рівні пріоритетів. Шістнадцять значень пріоритетів (16-31) відповідають групі пріоритетів реального часу, п'ятнадцять значень (1-15) призначені для звичайних потоків і значення 0 зарезервоване для системного потоку обнуління сторінок (див. рис. 4.2).

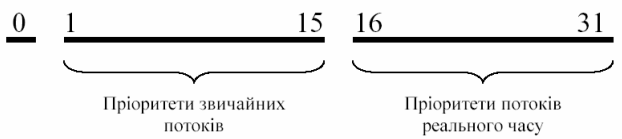


Рис. 4.2. Пріоритети потоків

Щоб позбавити користувача необхідності запам'ятовувати числові значення пріоритетів і мати можливість модифікувати планувальник, розробники ввели в систему *шар абстрагування пріо-*

ритетів. Наприклад, клас пріоритету для всіх потоків конкретного процесу можна задати за допомогою набору параметрів функції SetPriorityClass, які можуть мати такі значення:

- реального часу (REALTIME_PRIORITY_CLASS)
- високий (HIGH_PRIORITY_CLASS)
- вище за норму (ABOVE_NORMAL_PRIORITY_CLASS)
- нормальний (NORMAL_PRIORITY_CLASS)
- нижче від норми (BELOW_NORMAL_PRIORITY_CLASS)
- непрацюючий (IDLE_PRIORITY_CLASS).

Відносний пріоритет потоку встановлюється аналогічними параметрами функції SetThreadPriority.

Сукупність із шести класів пріоритетів процесів і семи класів пріоритетів потоків утворює 42 можливих комбінації і дозволяє сформувати так званий базовий пріоритет потоку (див. табл. 4.1).

Базовий пріоритет процесу і первинного потоку за замовчуванням дорівнює значенню із середини діапазонів пріоритетів процесів (24, 13, 10, 8, 6 або 4). Зміна пріоритету процесу спричиняє зміну пріоритетів усіх його потоків, при цьому їх відносні пріоритети залишаються без змін.

Пріоритети з 16 по 31 насправді пріоритетами реального часу не є, оскільки в рамках підтримки м'якого реального часу, яка реалізована в ОС Windows, ніяких гарантій щодо термінів виконання потоків не дається.

Таблиця 4.1

Формування базового пріоритету потоку з класу пріоритету процесу та відносного пріоритету потоку

	Пріоритети потоків						
Класи пріоритетів процесів	Критичний за часом	Найбільш високий	Вище за норму	Нормальний	Нижче від норми	Найбільш низький	Не працюючий
Непрацюючий	15	6	5	4	3	2	1
Нижче від норми	15	8	7	6	5	4	1
Нормальний	15	10	9	8	7	6	1
Вище за норму	15	12	11	10	9	8	1
Високий	15	15	14	13	12	11	1
Реального часу	31	26	25	24	23	22	16

Це просто вищі пріоритети, які зарезервовані для системних потоків і тих потоків, яким такий пріоритет дає користувач з адміністративними правами. Проте наявність пріоритетів реального часу, а також можливість витіснення коду ядра, локалізація сторінок пам'яті і ряд додаткових можливостей – все це дозволяє виконувати в середовищі ОС Windows програми м'якого реального часу, наприклад мультимедійні. Системний потік із нульовим пріоритетом займається зануленням сторінок пам'яті. Звичайні потоки користувача можуть мати пріоритети від 1 до 15.

Динамічне підвищення пріоритету

Планувальник ухвалює рішення на основі поточного пріоритету потоку, який може бути вищим за базовий. Є кілька ситуацій, коли має сенс підвищити пріоритет потоку.

Наприклад, після завершення операції введення-виведення збільшують пріоритет потоку, щоб дати йому можливість швидше почати виконання і, можливо, знов ініціювати операцію введення-виведення. У такий спосіб система заохочує інтерактивні потоки і підтримує зайнятість пристроїв введення-виведення. Величина, на яку підвищується пріоритет, не документована і залежить від пристрою (рекомендовані значення для диска і CD – це 1, для мережі – 2, клавіатури і миші – 6 і звукової карти – 8). Надалі протягом кількох квантів часу пріоритет плавно знижується до базового.

Іншими прикладами подібних ситуацій можуть служити: „пробудження" потоку після стану очікування семафора або іншої події; отримання потоком доступу до віконного введення.

Динамічне підвищення пріоритету розв'язує також проблему „голодування" потоків, які довго не одержували доступу до процесора. Виявивши такі потоки, що простоюють протягом приблизно 4 с, система тимчасово підвищує їх пріоритет до 15 і дає їм два кванти часу. Побічним наслідком застосування цієї технології може бути розв'язання відомої проблеми інверсії пріоритетів. Ця проблема виникає, коли низькопріоритетний потік утримує ресурс, блокуючи високопріоритетні потоки, що претендують на цей ресурс. Рішення полягає в штучному підвищенні його пріоритету на деякий час.

Динамічне підвищення пріоритетів покликане оптимізувати загальну пропускну спроможність системи, проте від нього виграють далеко не всі додатки. Виключення динамічного підвищення пріоритету можна здійснити за допомогою функцій SetProcessPriorityBoost і SetThreadPriorityBoost.

Дослідження програми, яка демонструє пріоритетне планування потоків

У прикладі 4.1 наведено текст програми, в якій три паралельні потоки A, B і C виконують тривалий однаковий рахунковий цикл. У програмі є можливість встановлення пріоритету того чи іншого потоку, як показано на рис. 4.3. Одразу ж після вибору пріоритету будь-якого потоку, програмний код потоків встановлюється у початковий стан та розпочинається їх виконання. Через різницю у пріоритетах потоків тривалі однакові рахункові цикли закінчаться неодночасно. Швидше за всіх будуть виконуватися програмні коди тих потоків, які мають найвищій пріоритет. З рис. 4.4 – 4.6 видно, що спочатку завершиться виконання потоку A, що має пріоритет вище від норми, через певний проміжок часу завершиться виконання потоку C, що має нормальний пріоритет, а останнім завершиться виконання потоку B, що має пріоритет нижче за норму. Якщо встановити функцією SetThreadPriority однакові пріоритети, то можна буде побачити, що всі три потоки закінчують роботу майже одночасно.

Як самостійну вправу рекомендується реалізувати гнучкіші сценарії планування, наприклад із додаванням функцій SwitchToThread (передача управління потоку) або Sleep (припинення потоку протягом заданого проміжку часу). Крім того, у MSDN є опис багатьох корисних функцій, пов'язаних з плануванням потоків.

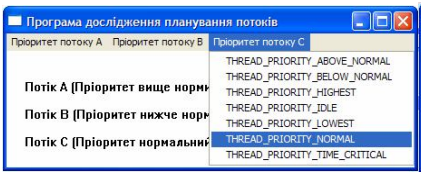


Рис. 4.3. Вибір пріоритетів потоків

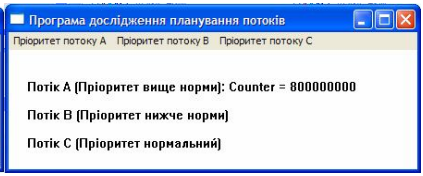


Рис. 4.4. Завершення роботи потоку A

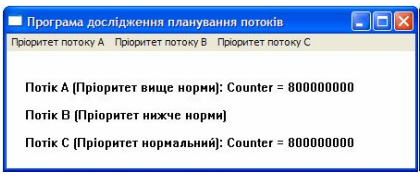


Рис. 4.5. Завершення роботи потоку C

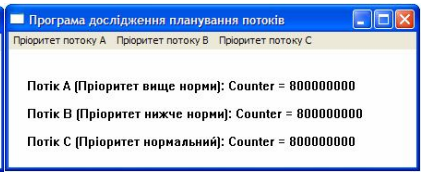


Рис. 4.6. Завершення роботи потоку B

Приклад 4.1

```
uses Windows,Messages;
{$r menu.res}
const MY_MENU = 10;
  UM_THREAD_DONE=WM_USER;
  IDM_TPAN_A = 100;
  IDM_TPBN_A = 101;
  IDM_TPH_A = 102;
  IDM_TPI_A = 103;
  IDM_TPL_A = 104;
  IDM_TPN_A = 105;
  IDM_TPTC_A = 106;
  IDM_TPAN_B = 200;
  IDM_TPBN_B = 201;
  IDM_TPH_B = 202;
  IDM_TPI_B = 203;
  IDM_TPL_B = 204;
  IDM_TPN_B = 205;
  IDM_TPTC_B = 206;
  IDM_TPAN_C = 300;
  IDM_TPBN_C = 301;
  IDM_TPH_C = 302;
  IDM_TPI_C = 303;
  IDM_TPL_C = 304;
  IDM_TPN_C = 305;
  IDM_TPTC_C = 306;
type ThreadManager =  packed record
    hwndParent: HWND;
    name: string;
```

```
    nValue:integer;  end;
var DC: hDC;
   ps:TPaintStruct;
   tex:array[1..3]of string;
   Tm:array[1..3]of ThreadManager;
   y,Threads_Prior:array[1..3]of integer;
   Handle_Threads:array[1..3]of HANDLE;
   i,j,t:integer;
procedure TR_PR_TEXT;
begin for j:=1 to 3 do begin
if Threads_Prior[j]=THREAD_PRIORITY_ABOVE_NORMAL  then
tex[j]:=tex[j]+' (Пріоритет вище за норму)';
if Threads_Prior[j]=THREAD_PRIORITY_BELOW_NORMAL  then
tex[j]:=tex[j]+' (Пріоритет нижче норми)';
if Threads_Prior[j]=THREAD_PRIORITY_HIGHEST       then
tex[j]:=tex[j]+' (Пріоритет найвищий)';
if Threads_Prior[j]=THREAD_PRIORITY_IDLE          then
tex[j]:=tex[j]+' (Пріоритет непрацюючий)';
if Threads_Prior[j]=THREAD_PRIORITY_LOWEST        then
tex[j]:=tex[j]+' (Пріоритет найнижчий)';
if Threads_Prior[j]=THREAD_PRIORITY_NORMAL        then
tex[j]:=tex[j]+' (Пріоритет нормальний)';
if Threads_Prior[j]=THREAD_PRIORITY_TIME_CRITICAL then
tex[j]:=tex[j]+' (Пріоритет критичний за часом)';
end;end;
procedure Thread_Sub_Func(count:dword);
begin     Tm[t].name:=tex[t];Tm[t].nValue:=count;
SendMessage(Tm[t].hwndParent,UM_THREAD_DONE,
                                          wParam(@Tm[t]),0);  end;
procedure ThreadFuncA;
var count:dword=0;
begin count:=0;for i:=1 to 800000000 do
inc(count);t:=1;Thread_Sub_Func(count);end;
procedure ThreadFuncB;
var count:dword=0;
```

```
begin count:=0;for i:=1 to 800000000 do
inc(count);t:=2;Thread_Sub_Func(count);end;
procedure ThreadFuncC;
var count:dword=0;
begin count:=0;for i:=1 to 800000000 do
inc(count);t:=3;Thread_Sub_Func(count);end;
function WindowProc conv arg_stdcall
(Window:HWND;Mess:UINT;Wp:WParam;Lp:LParam):LRESULT;
procedure Reset_Threads;
begin    for j:=1 to 3 do TerminateThread(Handle_Threads[j],0);
Handle_Threads[3]:=CreateThread(NIL,0,@ThreadFuncC,
                                                @Tm[3],0,hInstance);
Handle_Threads[2]:=CreateThread(NIL,0,@ThreadFuncB,
                                                @Tm[2],0,hInstance);
Handle_Threads[1]:=CreateThread(NIL,0,@ThreadFuncA,
                                                @Tm[1],0,hInstance);
tex[1]:='Потік A';tex[2]:='Потік B';tex[3]:='Потік C';
InvalidateRect(Window,nil,true);end;
begin
  case Mess of
     WM_CREATE:
     begin
     Tm[3].hwndParent:=Window;
     Handle_Threads[3]:=CreateThread(NIL,0,@ThreadFuncC,
                                                @Tm[3],0,hInstance);
     if (@Handle_Threads[3] = NIL)then MessageBox
               (Window,'Помилка створення потоку C',nil,MB_OK);
     Tm[2].hwndParent:=Window;
     Handle_Threads[2]:=CreateThread(NIL,0,@ThreadFuncB,
                                                @Tm[2],0,hInstance);
     if (@Handle_Threads[2] = NIL)then MessageBox
               (Window,'Помилка створення потоку B',nil,MB_OK);
     Tm[1].hwndParent:=Window;
     Handle_Threads[1]:=CreateThread(NIL,0,@ThreadFuncA,
                                                @Tm[1],0,hInstance);
     if (@Handle_Threads[1] = NIL)then MessageBox
```

```
                (Window,'Помилка створення потоку A',nil,MB_OK);
      TR_PR_TEXT;end;
      WM_COMMAND:
      begin
       case LoWord(Wp) of
          IDM_TPAN_A:begin              for j:=1 to 3 do
Threads_Prior[j]:=GetThreadPriority(Handle_Threads[j]);
Reset_Threads;Threads_Prior[1]:=
THREAD_PRIORITY_ABOVE_NORMAL;TR_PR_TEXT;
            for j:=1 to 3 do SetThreadPriority
                              (Handle_Threads[j],Threads_Prior[j]);end;
          IDM_TPBN_A:begin              for j:=1 to 3 do
Threads_Prior[j]:=GetThreadPriority(Handle_Threads[j]);
Reset_Threads;Threads_Prior[1]:=
THREAD_PRIORITY_BELOW_NORMAL;TR_PR_TEXT;
            for j:=1 to 3 do SetThreadPriority
                              (Handle_Threads[j],Threads_Prior[j]);end;
          IDM_TPH_A:begin              for j:=1 to 3 do
Threads_Prior[j]:=GetThreadPriority(Handle_Threads[j]);
            Reset_Threads;Threads_Prior[1]:=
THREAD_PRIORITY_HIGHEST;TR_PR_TEXT;
            for j:=1 to 3 do SetThreadPriority
                              (Handle_Threads[j],Threads_Prior[j]);end;
          IDM_TPI_A:begin              for j:=1 to 3 do
Threads_Prior[j]:=GetThreadPriority(Handle_Threads[j]);
            Reset_Threads;Threads_Prior[1]:=
THREAD_PRIORITY_IDLE;TR_PR_TEXT;
            for j:=1 to 3 do SetThreadPriority
                              (Handle_Threads[j],Threads_Prior[j]);end;
          IDM_TPL_A:begin              for j:=1 to 3 do
Threads_Prior[j]:=GetThreadPriority(Handle_Threads[j]);
            Reset_Threads;Threads_Prior[1]:=
THREAD_PRIORITY_LOWEST;TR_PR_TEXT;
            for j:=1 to 3 do SetThreadPriority
                              (Handle_Threads[j],Threads_Prior[j]);end;
```

```
        IDM_TPN_A:begin          for j:=1 to 3 do
Threads_Prior[j]:=GetThreadPriority(Handle_Threads[j]);
          Reset_Threads;Threads_Prior[1]:=
THREAD_PRIORITY_NORMAL;TR_PR_TEXT;
          for j:=1 to 3 do SetThreadPriority
                          (Handle_Threads[j],Threads_Prior[j]);end;
        IDM_TPTC_A:begin          for j:=1 to 3 do
Threads_Prior[j]:=GetThreadPriority(Handle_Threads[j]);
Reset_Threads;Threads_Prior[1]:=
THREAD_PRIORITY_TIME_CRITICAL;TR_PR_TEXT;
          for j:=1 to 3 do SetThreadPriority
                          (Handle_Threads[j],Threads_Prior[j]);end;
        IDM_TPAN_B:begin          for j:=1 to 3 do
Threads_Prior[j]:=GetThreadPriority(Handle_Threads[j]);
Reset_Threads;Threads_Prior[2]:=
THREAD_PRIORITY_ABOVE_NORMAL;TR_PR_TEXT;
          for j:=1 to 3 do SetThreadPriority
                          (Handle_Threads[j],Threads_Prior[j]);end;
        IDM_TPBN_B:begin          for j:=1 to 3 do
Threads_Prior[j]:=GetThreadPriority(Handle_Threads[j]);
Reset_Threads;Threads_Prior[2]:=
THREAD_PRIORITY_BELOW_NORMAL;TR_PR_TEXT;
          for j:=1 to 3 do SetThreadPriority
                          (Handle_Threads[j],Threads_Prior[j]);end;
        IDM_TPH_B:begin          for j:=1 to 3 do
Threads_Prior[j]:=GetThreadPriority(Handle_Threads[j]);
          Reset_Threads;Threads_Prior[2]:=
THREAD_PRIORITY_HIGHEST;TR_PR_TEXT;
          for j:=1 to 3 do SetThreadPriority
                          (Handle_Threads[j],Threads_Prior[j]);end;
        IDM_TPI_B:begin          for j:=1 to 3 do
Threads_Prior[j]:=GetThreadPriority(Handle_Threads[j]);
          Reset_Threads;Threads_Prior[2]:=
THREAD_PRIORITY_IDLE;TR_PR_TEXT;
          for j:=1 to 3 do SetThreadPriority
                          (Handle_Threads[j],Threads_Prior[j]);end;
```

```
        IDM_TPL_B:begin             for j:=1 to 3 do
Threads_Prior[j]:=GetThreadPriority(Handle_Threads[j]);
          Reset_Threads;Threads_Prior[2]:=
THREAD_PRIORITY_LOWEST;TR_PR_TEXT;
          for j:=1 to 3 do SetThreadPriority
                              (Handle_Threads[j],Threads_Prior[j]);end;
        IDM_TPN_B:begin             for j:=1 to 3 do
Threads_Prior[j]:=GetThreadPriority(Handle_Threads[j]);
          Reset_Threads;Threads_Prior[2]:=
THREAD_PRIORITY_NORMAL;TR_PR_TEXT;
          for j:=1 to 3 do SetThreadPriority
                              (Handle_Threads[j],Threads_Prior[j]);end;
        IDM_TPTC_B:begin            for j:=1 to 3 do
Threads_Prior[j]:=GetThreadPriority(Handle_Threads[j]);
Reset_Threads;Threads_Prior[2]:=
THREAD_PRIORITY_TIME_CRITICAL;TR_PR_TEXT;
          for j:=1 to 3 do SetThreadPriority
                              (Handle_Threads[j],Threads_Prior[j]);end;
        IDM_TPAN_C:begin            for j:=1 to 3 do
Threads_Prior[j]:=GetThreadPriority(Handle_Threads[j]);
Reset_Threads;Threads_Prior[3]:=
THREAD_PRIORITY_ABOVE_NORMAL;TR_PR_TEXT;
          for j:=1 to 3 do SetThreadPriority
                              (Handle_Threads[j],Threads_Prior[j]);end;
        IDM_TPBN_C:begin            for j:=1 to 3 do
Threads_Prior[j]:=GetThreadPriority(Handle_Threads[j]);
Reset_Threads;Threads_Prior[3]:=
THREAD_PRIORITY_BELOW_NORMAL;TR_PR_TEXT;
          for j:=1 to 3 do SetThreadPriority
                              (Handle_Threads[j],Threads_Prior[j]);end;
        IDM_TPH_C:begin             for j:=1 to 3 do
Threads_Prior[j]:=GetThreadPriority(Handle_Threads[j]);
          Reset_Threads;Threads_Prior[3]:=
THREAD_PRIORITY_HIGHEST;TR_PR_TEXT;
          for j:=1 to 3 do SetThreadPriority
                              (Handle_Threads[j],Threads_Prior[j]);end;
```

```
          IDM_TPI_C:begin              for j:=1 to 3 do
Threads_Prior[j]:=GetThreadPriority(Handle_Threads[j]);
           Reset_Threads;Threads_Prior[3]:=
THREAD_PRIORITY_IDLE;TR_PR_TEXT;
           for j:=1 to 3 do SetThreadPriority
                              (Handle_Threads[j],Threads_Prior[j]);end;
          IDM_TPL_C:begin              for j:=1 to 3 do
Threads_Prior[j]:=GetThreadPriority(Handle_Threads[j]);
           Reset_Threads;Threads_Prior[3]:=
THREAD_PRIORITY_LOWEST;TR_PR_TEXT;
           for j:=1 to 3 do SetThreadPriority
                              (Handle_Threads[j],Threads_Prior[j]);end;
          IDM_TPN_C:begin              for j:=1 to 3 do
Threads_Prior[j]:=GetThreadPriority(Handle_Threads[j]);
           Reset_Threads;Threads_Prior[3]:=
THREAD_PRIORITY_NORMAL;TR_PR_TEXT;
           for j:=1 to 3 do SetThreadPriority
                              (Handle_Threads[j],Threads_Prior[j]);end;
          IDM_TPTC_C:begin             for j:=1 to 3 do
Threads_Prior[j]:=GetThreadPriority(Handle_Threads[j]);
Reset_Threads;Threads_Prior[3]:=
THREAD_PRIORITY_TIME_CRITICAL;TR_PR_TEXT;
           for j:=1 to 3 do SetThreadPriority
                              (Handle_Threads[j],Threads_Prior[j]);end;
      end;      end;
     UM_THREAD_DONE:
     begin
     Str(Tm[t].nValue,tex[t]);
     tex[t]:=Tm[t].name+': Counter = '+tex[t];
     InvalidateRect(Window,nil,true);      end;
     WM_PAINT:
     begin      DC:= BeginPaint(Window, ps);
     for j:=1 to 3 do
     TextOut(DC,20,y[j],@tex[j]+1,length(tex[j]));
     EndPaint(Window, ps);
     end;
```

```
      WM_DESTROY:
      begin      for j:=1 to 3 do CloseHandle(Handle_Threads[j]);
      PostQuitMessage(0);      end;
  else Result := DefWindowProc(Window, Mess, Wp, Lp);
  end;
end;
var wc:TWndClass;wnd:HWnd;Msg:TMsg;
begin
y[1]:=30;y[2]:=60;y[3]:=90;
tex[1]:='Потік A';tex[2]:='Потік B';tex[3]:='Потік C';
  FillChar(wc, SizeOf(wc), 0);
  with wc do begin
    style:=CS_HREDRAW + CS_VREDRAW;
    lpfnWndProc := @WindowProc;
    cbClsExtra := 0;
    cbWndExtra := 0;
    hInstance := System.hInstance;
    hIcon := LoadIcon(THandle(NIL), IDI_APPLICATION);
    hCursor := LoadCursor(THandle(NIL), IDC_ARROW);
    hbrBackGround := GetStockObject(WHITE_BRUSH);
    lpszMenuName := MAKEINTRESOURCE(MY_MENU);
    lpszClassName := 'Лабораторна робота № 4';
end;
  if RegisterClass(wc) = 0 then Exit;
    wnd := CreateWindow(wc.lpszClassName,'Програма дослідження
планування потоків', WS_OVERLAPPEDWINDOW,
200,200,450,180,0,0,HInstance,nil);
  ShowWindow(wnd, SW_RESTORE);
  UpdateWindow(wnd);
  while GetMessage(Msg,0,0,0) do
  begin
    TranslateMessage(Msg);
    DispatchMessage(Msg);
  end;
end.
```

В цій програмі використовується файл ресурсів (menu.rc):

```
10 MENU DISCARDABLE
  BEGIN
  POPUP "&Пріоритет потоку A"
      BEGIN
      MENUITEM "&THREAD_PRIORITY_ABOVE_NORMAL",100
      MENUITEM "&THREAD_PRIORITY_BELOW_NORMAL",101
      MENUITEM "&THREAD_PRIORITY_HIGHEST",102
      MENUITEM "&THREAD_PRIORITY_IDLE",103
      MENUITEM "&THREAD_PRIORITY_LOWEST",104
      MENUITEM "&THREAD_PRIORITY_NORMAL",105
      MENUITEM "&THREAD_PRIORITY_TIME_CRITICAL",106
      END
  POPUP "&Пріоритет потоку B"
      BEGIN
      MENUITEM "&THREAD_PRIORITY_ABOVE_NORMAL",200
      MENUITEM "&THREAD_PRIORITY_BELOW_NORMAL",201
      MENUITEM "&THREAD_PRIORITY_HIGHEST",202
      MENUITEM "&THREAD_PRIORITY_IDLE",203
      MENUITEM "&THREAD_PRIORITY_LOWEST",204
      MENUITEM "&THREAD_PRIORITY_NORMAL",205
      MENUITEM "&THREAD_PRIORITY_TIME_CRITICAL",206
      END
  POPUP "&Пріоритет потоку C"
      BEGIN
      MENUITEM "&THREAD_PRIORITY_ABOVE_NORMAL",300
      MENUITEM "&THREAD_PRIORITY_BELOW_NORMAL",301
      MENUITEM "&THREAD_PRIORITY_HIGHEST",302
      MENUITEM "&THREAD_PRIORITY_IDLE",303
      MENUITEM "&THREAD_PRIORITY_LOWEST",304
      MENUITEM "&THREAD_PRIORITY_NORMAL",305
      MENUITEM "&THREAD_PRIORITY_TIME_CRITICAL",306
      END
  END
```

Величина кванта часу

Величина кванта часу має критичне значення для ефективної роботи системи в цілому. Необхідно зберегти інтерактивні якості системи і при цьому уникнути дуже частого перемикання контекстів. Імовірно оптимальне значення кванта (частка секунди) має забезпечувати обслуговування без перемикання запиту користувача, який займає процесор ненадовго, після чого зазвичай генерує запит на введення-виведення. В цьому випадку витрати на диспетчеризацію зводяться до мінімуму і забезпечуються прийнятні часи відгуків.

За замовчуванням початкова величина кванта в Windows Professional дорівнює двом інтервалам таймера, а у Windows Server ця величина збільшена до 12, щоб звести до мінімуму перемикання контексту. Тривалість інтервалу таймера визначається HAL і складає приблизно 10 мс для однопроцесорних x86 систем і 15 мс – для багатопроцесорних. Величину інтервалу системного таймера можна визначити за допомогою вільно поширюваної утиліти Clockres (сайт sysinternals.com).

Вибір між короткими і довгими значеннями можна зробити за допомогою панелі "Властивості" ("Мій комп'ютер"). Величина кванта задається в параметрі HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl\Win32PrioritySeparation реєстру.

Планування в умовах багатопроцесорності

Реєнтерабельність коду ядра дозволяє ОС Windows підтримувати симетричні мультипроцесорні системи (процесори ідентичні). Необхідність завантаження декількох процесорів ускладнює завдання планування. Кількість процесорів система визначає при завантаженні, і ця інформація стає доступною додаткам через функцію GetSystemInfo. Кількість процесорів, використовуваних системою, може бути обмежена за допомогою параметра *NumPcs* із файла Boot.ini.

Ведення окремих черг готових до виконання потоків для кожного з процесорів може мати наслідком нерівномірне завантаження процесорів, тому використовується загальна черга потоків у стані готовності. Будь-який потік стає в чергу і планується на

будь-який доступний процесор. Оскільки в системі немає головного процесора, кожен процесор займається самоплануванням і вибирає потік із черги готовності. Щоби гарантувати, що два процесори не виберуть один і той самий потік, для кожного процесора організується ексклюзивний доступ до даної черги за рахунок використання спін-блокування диспетчера ядра.

Прив'язка до процесорів

У кожного потоку є *маска прив'язки до процесорів* (affinity mask), яка вказує, на яких процесорах можна виконувати даний потік. За замовчуванням Windows використовує нежорстку прив'язку (soft affmity) потоків до процесорів. Це означає, що деяку перевагу має останній процесор, на якому виконувався потік, щоб повторно використовувати дані з кеша цього процесора (споріднене планування). Потоки успадковують маску прив'язки процесу. Зміна прив'язки процесу і потоку може бути здійснена за допомогою Win32-функцій SetProcessAffinityMask і SetThreadAfftnityMask або за допомогою інструментальних засобів Windows (наприклад, це може зробити диспетчер завдань). Є також можливість сформувати апріорну маску прив'язки у файлі образі процесу, що запускається.

Окрім номера останнього процесора, в блоці ядра потоку KTHREAD зберігається номер *ідеального процесора* (ideal processor) – переважаючого для виконання даного потоку. Ідеальний процесор вибирається випадковим чином при створенні потоку. Це значення збільшується на 1 всякий раз, коли створюється новий потік, тому створювані потоки рівномірно розподіляються по набору доступних процесорів. Потік може змінити це значення за допомогою функції SetThreadIdealProcessor.

Готовий до виконання потік система намагається підключити до простоюючого процесора. Якщо таких кілька, то перевага віддається ідеальному процесорові даного потоку, а потім останньому з процесорів, на якому потік виконувався. Якщо всі процесори зайняті, то робиться перевірка на можливість витіснити якийнебудь потік, що виконується або чекає (насамперед на ідеальному процесорі, потім – на останньому для даного потоку). Якщо витіснення неможливе, новий потік поміщається в чергу готових

потоків із відповідним рівнем пріоритету і чекає виділення процесорного часу.

Отже, в ОС Windows реалізоване дворівневе планування. На верхньому рівні алгоритму потоки приписуються конкретним (ідеальним, останнім, найменш завантаженим) центральним процесорам, унаслідок чого в кожного процесора створюється своя черга потоків. На нижньому рівні кожним процесором здійснюється реальне планування за допомогою пріоритетів та інших засобів.

Наприклад, якщо який-небудь процесор починає простоювати, в завантаженого роботою процесора відбирається потік і віддається йому. Дворівневе планування рівномірно розподіляє навантаження серед процесорів і використовує перевагу спорідненості кеша.

Жорстка прив'язка (hard affinity), що виконується за допомогою функцій SetProcessAffinityMask і SetThreadAfftnityMask, доцільна в архітектурі з доступом, що неуніфікується (NUMA), де швидкість доступу до пам'яті залежить від взаємного розташування процесорів і банків пам'яті на системній платі.

Таким чином, процесорний час – обмежений ресурс, тому планування – важлива і критична для продуктивності операція. Одне з ключових питань – вибір моменту для запуску процедури планування. У системі реалізовано пріоритетне витісняюче планування з динамічними пріоритетами. Для зручності користувача і мобільності програм підтримується шар абстрагування пріоритетів. Механізми прив'язки дозволяють організувати ефективне виконання програм у багатопроцесорних системах.

Практична частина

1. Складіть програму, наведену у прикладі 4.1 теоретичної частини даної лабораторної роботи. Запустіть програму та переконайтеся, що вона працює правильно (згідно з описом алгоритму її роботи).

2. Оформіть звіт із виконаної лабораторної роботи, який повинен містити текст програми, демонстрацію результатів роботи, та дайте відповіді на контрольні запитання.

Контрольні запитання та завдання

1. Що собою являє процес планування потоків у операційних системах?
2. Чим викликається запуск процедури планування?
3. Які алгоритми планування ви знаєте? Опишіть їх.
4. Що таке реєнтерабельність програмного коду ядра операційної системи?
5. Які класи пріоритетів процесів та відносні пріоритети потоків ви знаєте?
6. У яких випадках виконується динамічне підвищення пріоритету та як його виключити?
7. Які критерії вибору величини кванта часу?
8. Опишіть особливості планування в умовах багатопроцесорності.
9. Що таке маска прив’язаності до процесорів?

Лабораторна робота № 5
Дослідження способів організації обміну інформацією між процесами в операційній системі Windows

Теоретична частина

Вступ

З лекційного курсу „Операційні системи" відомо, що для виконання таких завдань, як сумісне використання даних, побудова інтегрованих багатофункціональних додатків і т.д., різним процесам (а також різним потокам) необхідно взаємодіяти між собою. Оскільки процеси можна розглядати як відособлені один від одної сутності, для забезпечення коректної взаємодії процесів потрібні спеціальні засоби і дії операційної системи.

Відомо також, що в основі міжпроцесного (Inter Process Communications, IPC) обміну зазвичай знаходиться розподілений ресурс (наприклад, канал або сегмент розподіленої пам'яті), а отже, ОС повинна надати засоби для генерації, іменування, установки режиму доступу і атрибутів захисту таких ресурсів. Здебільшого такий ресурс може бути доступний усім процесам, які знають його ім'я і мають необхідні привілеї.

Крім того, організація зв'язку між процесами завжди припускає встановлення таких її характеристик, як:

- напрям зв'язку. Зв'язок однонаправлений (симплексний) і двонаправлений (напівдуплексний для почергової передачі інформації і дуплексний із можливістю одночасної передачі даних у різних напрямах);
- тип адресації. У випадку прямої адресації інформація посилається безпосередньо одержувачу, наприклад процесу P-Send (P, message). У випадку непрямої або опосередкованої адресації інформація поміщається в деякий проміжний об'єкт, наприклад у поштову скриньку;
- використовувана модель передачі даних – потокова або модель повідомлень (див. нижче);
- об'єм інформації, що передається, і відомості про те, чи володіє канал буфером необхідного розміру;

- синхронність обміну даними. Якщо відправник повідомлення блокується до отримання цього повідомлення адресатом, то обмін вважається синхронним, інакше – асинхронним.

Окрім перерахованих, у кожного зв'язку є ще ряд особливостей.

Способи міжпроцесного обміну

Традиційно вважається, що основними способами міжпроцесного обміну є канали і розподілена пам'ять (рис 5.1), які базуються на відповідних об'єктах ядра.

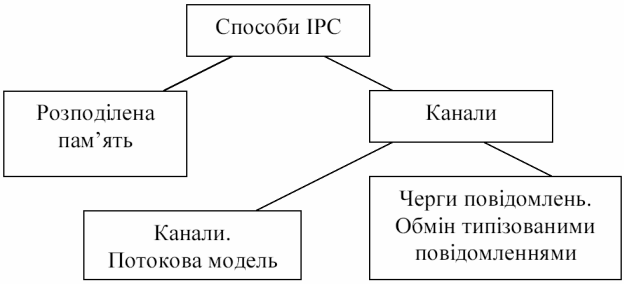


Рис. 5.1. Основні способи міжпроцесного обміну

У випадку розподіленої пам'яті два або більше процесів спільно використовують сегмент пам'яті. Спілкування відбувається за допомогою звичайних операцій копіювання або переміщення даних у пам'яті (засобами звичайних мов програмування).

Канали припускають створені засобами операційної системи лінії зв'язку. Двома основними моделями передачі даних по каналу є потік введення-виведення і повідомлення. При передачі в рамках потокової моделі дані є неструктурованою послідовністю байтів і ніяк не інтерпретуються системою. У моделі повідомлень на дані, які передаються, накладається деяка структура, зазвичай їх розділяють на повідомлення наперед обумовленого формату.

Є також й інші механізми міжпроцесного обміну, реалізовані в ОС Windows, наприклад сокети, Clipboard або віддалений виклик процедури (RPC). Вичерпна довідкова інформація на цю тему є у відповідній літературі.

Поняття про розподілений ресурс

Міжпроцесний обмін базується на розподілених ресурсах, до яких має доступ деяка множина процесів. При цьому виникають завдання створення, іменування і захисту таких ресурсів. Зазвичай один із процесів створює ресурс, наділяє його атрибутами захисту і ім'ям, по якому даний ресурс може бути доступний решті процесів (навіть у разі завершення роботи процесу-творця).

Як приклад розглянемо спілкування через розподілену пам'ять (рис. 5.2).

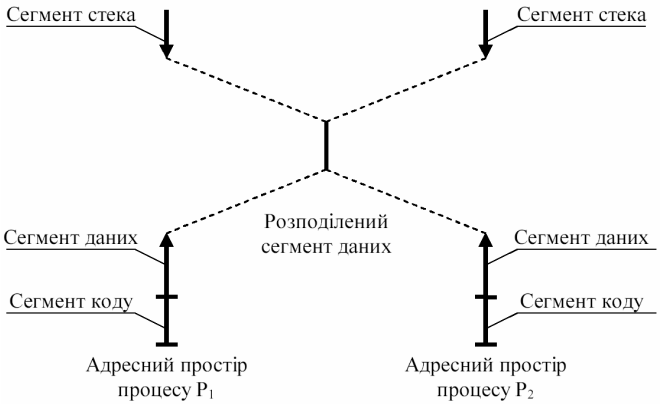


Рис. 5.2. Адресні простори процесів, що взаємодіють через сегмент пам'яті, що розділяється

У ОС Windows сегмент розподіленої пам'яті створюється за допомогою Win32-функції CreateFileMapping (див. рис. 5.3). У випадку успішного виконання даної функції створюється ресурс – фрагмент пам'яті, доступний по імені (параметр lpname), який базується на відповідному об'єкті ядра – "об'єкті-файлі, що відображається в пам'ять" із властивими будь-якому об'єктові атрибутами. Процесу-творцю повертається описувач (handle) ресурсу. Інші процеси, охочі мати доступ до ресурсу, також повинні одержати його описувач. У даному випадку це можна зробити за допомогою функції OpenFileMapping, вказавши ім'я ресурсу як одного з параметрів.

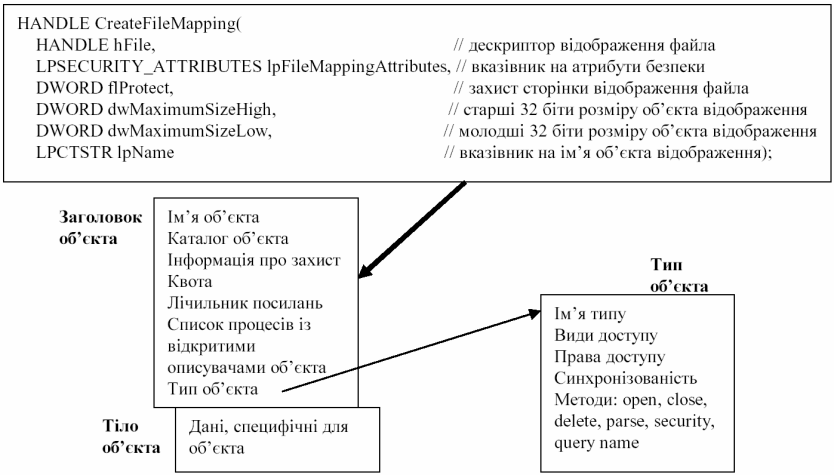


Рис. 5.3. Створення сегмента розподіленої пам'яті базується на розподіленому ресурсі, якому відповідає об'єкт ядра

Способи створення і характеристики файлів, що відображаються в пам'ять, будуть розглянуті в частині курсу "Система управління пам'яттю", а в рамках даної лабораторної роботи обмежимося відомостями про обмін інформації по каналах зв'язку. При цьому не треба забувати, що при будь-якому способі спілкування в рамках однієї обчислювальної системи завжди використовуватиметься елемент загальної пам'яті. Інша справа, що у випадку каналів ця пам'ять може бути виділена не в адресному просторі процесу, а в адресному просторі ядра системи, як це показано на рис. 5.4.

Дослідження роботи програм, що імітують функції сервера і клієнта

Для того, щоб показати використання механізмів обміну між процесами за допомогою розподіленої пам'яті (файла, відображеного в пам'ять) та за допомогою повідомлення WM_COPYDATA, ми розробимо дві програми, які імітують функції сервера і клієнта. *Клієнтом* називається об'єкт, який запитує доступ до служби або ресурсу. *Сервер* – це об'єкт, який виконує деяку службу або володіє ресурсом.

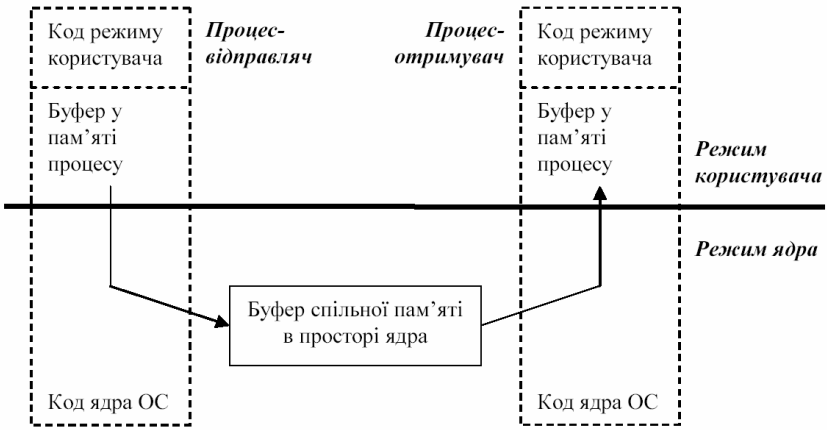


Рис. 5.4. Обмін через канали зв'язку здійснюється через буфер в адресному просторі ядра системи

Клієнт і сервер можуть працювати на одній і тій самій машині, використовуючи локальні механізми комунікації, або на різних машинах, використовуючи для зв'язку мережні засоби.

Поведінка клієнта та сервера асиметрична. Процес-сервер ініціюється і переходить у стан очікування запитів від можливих клієнтів. Як правило, процес-клієнт запускається в інтерактивному режимі й посилає запити серверу. Сервер виконує отриманий запит, причому це може передбачати діалог із клієнтом, а може не передбачати. Потім сервер знову переходить у стан очікування запитів від інших клієнтів.

У прикладах 5.1, 5.2, які наведені нижче, клієнтом і сервером є додатки ClientApp і ServerApp. На рис. 5.5 показано, як здійснюється взаємозв'язок між цими додатками через локальні механізми комунікації.

Хоча на рисунку зображений лише один клієнт, насправді їх може бути кілька. Максимально можлива кількість клієнтів визначається „потужністю" або пропускною здатністю сервера.

Сценарій взаємодії клієнта і сервера передбачає початкову фазу, в якій клієнт посилає серверу запит на зв'язок і отримує відповідь-повідомлення як ознаку встановлення зв'язку. Після цього починається робоча фаза, в якій із деякою періодичністю клієнт

записує свій робочий запит у розподілену пам'ять і чекає відповіді від сервера. Ця відповідь також записується розподілену пам'ять. Синхронізація взаємодії клієнта та сервера здійснюється через об'єкти ядра „події". Отже, в робочій фазі повторюються дії, помічені на рис. 5.5 номерами 3 – 6.

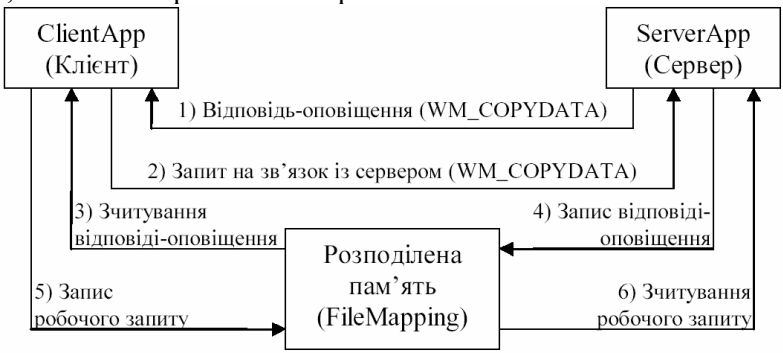


Рис. 5.5. Взаємодія клієнта та сервера

Обмін даними за допомогою повідомлення WM_COPYDATA

Системне повідомлення WM_COPYDATA є, імовірно, найбільш простим методом обміну даними між процесами.

Щоб послати будь-яке повідомлення за допомогою функції SendMessage, необхідно знати дескриптор вікна, якому воно посилається. Зазвичай цей дескриптор отримують викликом функції FindWindow, наприклад:

```
hwdServer := FindWindow(nil,'ServerApp');.
```

За допомогою цієї інструкції клієнтський додаток розпізнає дескриптор основного вікна серверного додатка, який має заголовок ServerApp.Перш ніж посилати повідомлення WM_COPYDATA, ви повинні визначити структурну змінну типу COPYDATASTRUCT. Ця структура визначена так:

```
typedef struct tagCOPYDATASTRUCT {
   DWORD dwData;
   DWORD cbData;
   PVOID lpData;
} COPYDATASTRUCT;.
```

В поле dwData можна записати 32-бітне число, яке буде передане додаткові-отримувачу.

Поле lpData може містити вказівник на дані, які передаються додатку-отримувачу. Якщо lpData не дорівнює nil, то поле cbData повинно містити розмір у байтах даних, на які вказує lpData.

В залежності від потреб додатка ви можете використовувати обидва вказаних поля (dwData і lpData) або тільки одне з них.

Коли ви посилаєте повідомлення WM_COPYDATA за допомогою функції SendMessage, адреса змінної типу COPYDATASTRUCT повинна передаватися в параметрі lParam.

Наприклад, нижчеподані інструкції у клієнтському додаткові підготовлюють запит у вигляді рядка request і посилають його серверному додатку, який має заголовок вікна ServerApp:

```
request:='ClientApp';
cds.lpData:=@request;
cds.cbData:=Length(request)+1;
hwndServer:=FindWindow(nil,'ServerApp');
SendMessage(hwndServer,WM_COPYDATA,Window,Longint(@cds));.
```

Програма, якій адресується повідомлення WM_COPYDATA, повинна розглядати дані, на які вказує lpData, як дані лише для зчитування. Вказівник lpData коректний лише в процесі обробки повідомлення WM_COPYDATA. Якщо додаток, який отримує повідомлення, має намір використовувати отримані дані в подальшій своїй роботі, то він повинен скопіювати їх у локальний буфер. Дані, розміщені за адресою lpData, не повинні містити жодних вказівників, оскільки в адресному просторі додатка, який приймає, ці вказівники будуть інтерпретуватися неправильно.

Текст програми додатка ServerApp наведено у прикладі 5.1

Приклад 5.1

```
uses Windows,Messages,Strings;
const MAX_N=10;
type ThreadManager =  packed record
    hwndParent: HWND;end;
var i,N,k,dw:dword:=0;
   hEvtRecToServ:array[0..MAX_N]of handle;
   eventName:array[0..MAX_N]of string;
```

```
    hThread,hEvtServIsExist,hEvtServIsFree,hEvtServIsDone,
                                                    hFileMap:handle;
    tm:ThreadManager;
    request,suffix,tex:string;
    pView:PVOID;
    hwndClient,hParent:HWND;
    DC: hDC;
    ps:TPaintStruct;
    cds:tCOPYDATASTRUCT;
    pcds:^tCOPYDATASTRUCT;
procedure ThreadFunc;
begin
hParent:=tm.hwndParent;
while true do begin
dw:=WaitForMultipleObjects
                          (MAX_N,@hEvtRecToServ,false,INFINITE);
case dw of
WAIT_FAILED:MessageBox(hParent,'Помилка виклику
WaitForMultipleObjects','ServerApp',MB_OK);
else begin
k:=dw-WAIT_OBJECT_0;
pView:=MapViewOfFile(hFileMap,FILE_MAP_WRITE,0,0,0);
strCopy(@tex,pView);
suffix:=' - '+eventName[k];
tex:=tex+suffix;
strCopy(pView,@tex);
UnmapViewOfFile(pView);
SetEvent(hEvtServIsDone);
    end;
end;
          end;
end;
function WindowProc conv arg_stdcall
(Window:HWND;Mess:UINT;Wp:WParam; Lp:LParam):LRESULT;
begin
  case Mess of
```

```
    WM_CREATE:
    begin
hEvtServIsExist:=OpenEvent(EVENT_ALL_ACCESS,false,'Server');
    if hEvtServIsExist<>0 then begin MessageBox(Window,'Сервер
вже виконується. Другий екземпляр заборонено','ServerApp',
                MB_OK+MB_ICONSTOP+MB_SYSTEMMODAL);
                    PostQuitMessage(0);
                    end else
hEvtServIsExist:=CreateEvent(nil,false,false,'Server');
    end;
    WM_COPYDATA:
    begin
    if N=MAX_N then MessageBox(Window,'Сервер перевантаже-
ний. В доступі відмовлено','ServerApp',
                MB_OK+MB_ICONSTOP+MB_SYSTEMMODAL);
    pcds:=pointer(Lp);
    StrLCopy(@request,pcds^.lpData,pcds^.cbData);
    str(N,suffix);eventName[N]:=request+'_'+suffix;
    hEvtRecToServ[N]:=CreateEvent(nil,false,false,@eventName[N]);
    if N=0 then begin
            for i:=0 to MAX_N do
hEvtRecToServ[i]:=hEvtRecToServ[0];
hFileMap:=CreateFileMapping(INVALID_HANDLE_VALUE,nil,
                    PAGE_READWRITE, 0,4*1024,'SharedData');
hEvtServIsFree:=CreateEvent(nil,false, true,'ServerIsFree');
hEvtServIsDone:=CreateEvent(nil,false,false,'ServerIsDone');
            tm.hwndParent:=Window;
hThread:=CreateThread(NIL,0,@ThreadFunc,@tm,0,hInstance);
            end;
    hwndClient:=FindWindow(nil,'ClientApp');
    cds.dwData:=N;
    cds.lpData:=@eventName[N];
    cds.cbData:=length(eventName[N])+1;
SendMessage(hwndClient,WM_COPYDATA,
                                        Window,Longint(@cds));
    if N<MAX_N then inc(N);
```

```
      InvalidateRect(Window,nil,true);       end;
      WM_PAINT:
      begin      DC:= BeginPaint(Window, ps);
      if N>0 then begin
      str(N,tex);tex:='Кількість клієнтів, які обслуговуються: '+tex;
      TextOut(DC,10,20,@tex+1,length(tex));                  end;
      EndPaint(Window, ps);       end;
      WM_DESTROY:
      begin      CloseHandle(hFileMap);
      PostQuitMessage(0);       end;
  else Result := DefWindowProc(Window, Mess, Wp, Lp);
  end; end;
var wc:TWndClass;wnd:HWnd;Msg:TMsg;
begin
  FillChar(wc, SizeOf(wc), 0);
  with wc do begin
    style:=CS_HREDRAW + CS_VREDRAW;
    lpfnWndProc := @WindowProc;
    cbClsExtra := 0;
    cbWndExtra := 0;
    hInstance := System.hInstance;
    hIcon := LoadIcon(THandle(NIL), IDI_APPLICATION);
    hCursor := LoadCursor(THandle(NIL), IDC_ARROW);
    hbrBackGround := GetStockObject(WHITE_BRUSH);
    lpszMenuName := nil;
    lpszClassName := 'Лабораторна робота № 5. Сервер';end;
  if RegisterClass(wc) = 0 then Exit;
    wnd := CreateWindow(wc.lpszClassName,'ServerApp',
    WS_OVERLAPPEDWINDOW,200,200,450,80,0,0,HInstance,nil);
  ShowWindow(wnd, SW_RESTORE);
  UpdateWindow(wnd);
  while GetMessage(Msg,0,0,0) do
  begin
    TranslateMessage(Msg);
    DispatchMessage(Msg);
  end;  end.
```

Після старту сервер чекає першого запиту на зв'язок від можливого клієнта. Цей запит надходить у вигляді повідомлення WM_COPYDATA. В коді програми ClientApp, який наводиться нижче, можна побачити, що клієнт посилає в цьому повідомленні рядок, який містить його ім'я (воно збігається із заголовком головного вікна додатка). Сервер витягує цей рядок request і додає до нього суфікс, що містить символ підкреслення і номер запиту N. Модифікований рядок запам'ятовується в елементі масиву eventName[N]. Потім сервер створює об'єкт-подію з дескриптором hEvtRecToServ[N] та ім'ям eventName[N].

Якщо запит, що надійшов, перший і N має нульове значення, то сервер виконує ряд ініціюючих дій:

- Всі елементи масиву hEvtRecToServ[N] заповнюються значеннями дескриптора hEvtRecToServ[0]. Якщо цього не зробити, то функція WaitForMultipleObjects, яка викликається пізніше для масиву подій hEvtRecToServ, працювати не буде.

- Створюється об'єкт „проекція файла" з дескриптором hFileMap та ім'ям SharedData. При цьому розмір файла, що задається четвертим і п'ятим параметрами функції CreateFileMapping, дорівнює 4 Кбайт. Ця величина повинна бути кратна розміру сторінки пам'яті. А третій параметр визначає, що проекція файла буде використовуватися і для запису, і для зчитування.

- Створюються об'єкти-події hEvtServIsFree та hEvtServIsDone.

- Запускається вторинний потік із вхідною функцією ThreadFunc. Цей потік призначений для обробки „робочих запитів" клієнтів.

Завершуючи обробку прийнятого повідомлення WM_COPYDATA, сервер відправляє ім'я eventName[N] назад клієнту, використовуючи для цього все те ж повідомлення WM_COPYDATA. Це повідомлення є відповіддю-оповіщенням для клієнта (див. рис. 5.5).

Алгоритм функції потоку ThreadFunc досить простий. Як тільки сервер виявляє звільнення якої-небудь події з масиву hEvtRecToServ, він визначає індекс клієнта і приступає до обробки „робочого запиту". Зверніть увагу на те, що після запису „відповіді" в розподілену пам'ять потік викликає функцію SetEvent, щоб пе-

ревести подію hEvtServIsDone у вільний стан. Як ви побачите нижче, клієнтський додаток буде очікувати на цю подію, щоб отримати відповідь сервера з розподіленої пам'яті.

Програмний код додатка ClientApp наведено у прикладі 5.2.

Створивши додаток, додайте до нього ресурс меню з ідентифікатором MENU. Меню повинно містити один пункт з ім'ям „***З'єднатися із сервером***" та ідентифікатором IDM_LINK.

Приклад 5.2

```
uses Windows,Messages,Strings;
{$r menu.res}
const MY_MENU =
10;IDM_LINK=100;TIMER_ID=1;TIMER_PERIOD=500;
var hEvtRecToServ,hEvtServIsFree,hEvtServIsDone,hFileMap:handle;
   pView:PVOID;
   hwndServer:HWND;
   DC: hDC;
   ps:TPaintStruct;
   cds:tCOPYDATASTRUCT;
   pcds:^tCOPYDATASTRUCT;
   isLinkToServer,bServerIsDone:boolean;
   i,dw0,dw1,count:dword:=0;
   request,eventName,tex,msgSended,msgReceived:string;
function WindowProc conv arg_stdcall
(Window:HWND;Mess:UINT;Wp:WParam; Lp:LParam):LRESULT;
begin
 case Mess of
     WM_CREATE:
     begin      isLinkToServer:=false;
     msgSended:=‘‘;
     msgReceived:=‘‘;      end;
     WM_COMMAND:
     begin
      case LoWord(Wp) of
      IDM_LINK:begin
            cds.dwData:=0;
            request:=‘ClientApp’;
```

```
            cds.lpData:=@request;
            cds.cbData:=Length(request)+1;
            hwndServer:=FindWindow(nil,'ServerApp');
            if hwndServer=0 then MessageBox(Window,'Помилка
зв''язку','ClientApp',MB_OK);
SendMessage(hwndServer,WM_COPYDATA,
                                                Window,Longint(@cds));
            end;      end;      end;
      WM_COPYDATA:
      begin      pcds:=pointer(Lp);
      StrLCopy(@eventName,pcds^.lpData,pcds^.cbData);
hEvtRecToServ:=OpenEvent(EVENT_ALL_ACCESS,false,
                                                @eventName);
hEvtServIsFree:=OpenEvent(EVENT_ALL_ACCESS,false,
                                                'ServerIsFree');
hEvtServIsDone:=OpenEvent(EVENT_ALL_ACCESS,false,
                                                'ServerIsDone');
hFileMap:=OpenFileMapping(FILE_MAP_ALL_ACCESS,false,
                                                'SharedData');
 for i:=1 to length(eventName)do tex[i-1]:=eventName[i];tex[i]:=chr(0);
      SetWindowText(Window,@tex);
      isLinkToServer:=true;
      InvalidateRect(Window,nil,true);
      SetTimer(Window,TIMER_ID,TIMER_PERIOD,nil);
      end;
      WM_TIMER:
      begin
      dw0:=WaitForSingleObject(hEvtServIsFree,TIMER_PERIOD);
         case dw0 of
         WAIT_OBJECT_0:begin
        pView:=MapViewOfFile(hFileMap,FILE_MAP_WRITE,0,0,0);
         inc(count);str(count,tex);
         strCopy(pView,@tex);
         msgSended:='Запит до сервера:          '+tex;
         UnmapViewOfFile(pView);
         InvalidateRect(Window,nil,false);
```

```
        SetEvent(hEvtRecToServ);
      dw1:=WaitForSingleObject(hEvtServIsDone,TIMER_PERIOD);
          case dw0 of
          WAIT_OBJECT_0:begin
        pView:=MapViewOfFile(hFileMap,FILE_MAP_READ,0,0,0);
          strCopy(@tex,pView);
          msgReceived:='Відповідь від сервера:        '+tex;
          UnmapViewOfFile(pView);
          bServerIsDone:=true;
          SetEvent(hEvtServIsFree);
          InvalidateRect(Window,nil,false);          end;
          WAIT_TIMEOUT:begin end;
          WAIT_FAILED:MessageBox(Window,'Помилка очіку-
вання hEvtServIsDone','ClientApp',MB_OK);
          end;
        end;
        WAIT_TIMEOUT:begin end;
        WAIT_FAILED:MessageBox(Window,'Помилка очікування
hEvtServIsFree','ClientApp',MB_OK);
        end;
    end;
    WM_PAINT:
    begin
    DC:= BeginPaint(Window, ps);
    if isLinkToServer then begin
tex:='Встановлено зв''язок із сервером через подію '+eventName;
    TextOut(DC,20,20,@tex+1,length(tex));
    TextOut(DC,20,40,@msgSended+1,length(msgSended));
    end;
    if bServerIsDone then begin
    TextOut(DC,20,60,@msgReceived+1,length(msgReceived));
    end;
    EndPaint(Window, ps);      end;
    WM_DESTROY:
    begin
    UnmapViewOfFile(pView);
```

```
      CloseHandle(hFileMap);
      PostQuitMessage(0);       end;
  else Result := DefWindowProc(Window, Mess, Wp, Lp);
  end;
end;
var wc:TWndClass;wnd:HWnd;Msg:TMsg;
begin
  FillChar(wc, SizeOf(wc), 0);
  with wc do begin
    style:=CS_HREDRAW + CS_VREDRAW;
    lpfnWndProc := @WindowProc;
    cbClsExtra := 0;
    cbWndExtra := 0;
    hInstance := System.hInstance;
    hIcon := LoadIcon(THandle(NIL), IDI_APPLICATION);
    hCursor := LoadCursor(THandle(NIL), IDC_ARROW);
    hbrBackGround := GetStockObject(WHITE_BRUSH);
    lpszMenuName := MAKEINTRESOURCE(MY_MENU);
    lpszClassName := 'Лабораторна робота № 5. Клієнт';end;
  if RegisterClass(wc) = 0 then Exit;
    wnd := CreateWindow(wc.lpszClassName,'ClientApp',
   WS_OVERLAPPEDWINDOW,200,200,450,140,0,0,HInstance,nil);
  ShowWindow(wnd, SW_RESTORE);
  UpdateWindow(wnd);
  while GetMessage(Msg,0,0,0) do
  begin
    TranslateMessage(Msg);
    DispatchMessage(Msg);
  end;
end.
```

На рис. 5.6 показано в роботі сервер ServerApp та три клієнти ClientApp. Коли додаток запущений, робота клієнта починається після того, як користувач виконає команду меню „*З'єднатися із сервером*". Обробляючи цю команду (case IDM_LINK), програма посилає серверу повідомлення WM_COPYDATA, яка містить вказівник на рядок request.

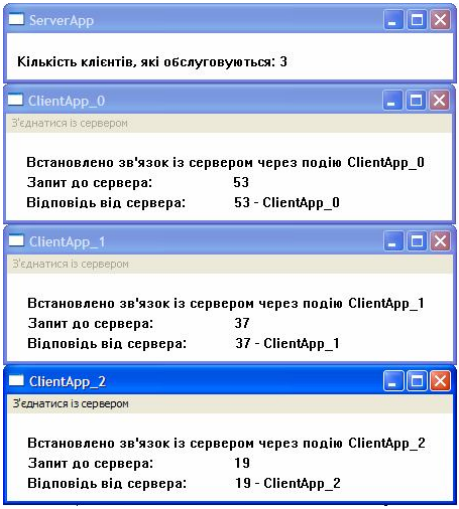


Рис. 5.6. Сервер ServerApp обслуговує трьох клієнтів ClientApp

Отримавши відповідь-оповіщення від сервера також через повідомлення WM_COPYDATA, клієнт виконує такі дії:

- отримує ім’я розподіленого об’єкта-події eventName;
- відкриває розподілені об’єкти-події, отримуючи дескриптори hEvtRecToServ, hEvtServIsFree, hEvtServIsDone (в подальшому клієнт використовує подію hEvtRecToServ, щоб інформувати сервер про готовність „робочого запиту”);
- відкриває існуючий об’єкт „проекція файла” з ім’ям SharedData (сервер повинен бути запущений раніше від клієнта);
- замінює текст заголовка свого вікна рядком eventName;
- запускає стандартний таймер із періодом TIMER_PERIOD.

Далі робота програми керується перериваннями від таймера. Обробляючи повідомлення WM_TIMER, додаток за допомогою функції WaitForSingleObject перевіряє, чи вільний сервер. Якщо так, то в розподілену пам’ять записується „робочий запит”, який являє собою рядок із номером запиту. Якщо ні, то додаток чекає звільнення сервера. Після успішного запису „робочого запиту” в розподілену пам’ять програма переводить у вільний стан об’єкт-подію hEvtRecToServ. Саме цю подію очікує сервер. Потім клієнт переходить до очікування події hEvtServIsDone, яка свідчить про

те, що сервер виконав запит. Як тільки вказана подія звільняється, отримана від сервера відповідь отримується з розподіленої пам'яті і виводиться у вікно додатка.

Канали зв'язку

Основний принцип роботи каналу полягає в буферизації виведення одного процесу і забезпеченні можливості читання вмісту програмного каналу іншим процесом. При цьому інтерфейс програмного каналу часто збігається з інтерфейсом звичайного файла і реалізується звичайними файловими операціями read і write. Для обміну можуть використовуватися потокова модель і модель обміну повідомленнями.

Механізм генерації каналу припускає отримання процесом-творцем (процесом-сервером) двох описувачів (handles) для користування цим каналом. Один з описувачів застосовується для читання з каналу, інший – для запису в канал.

Один з варіантів використання каналу – це його використання процесом для взаємодії з самим собою. Розглянемо наступне зображення системи, що складається з процесу і ядра, після створення каналу (рис 5.7):

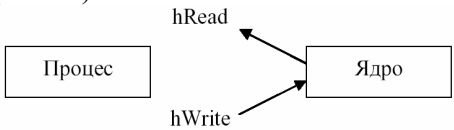


Рис. 5.7. Спілкування процесу із самим собою через канал зв'язку

Із цього рисунка легко побачити, що навіть якщо процес посилає дані самому собі, вони проходять через ядро. Отже, для організації таких каналів, а також їх іменування в ядрі повинні бути реалізовані елементи файлової системи.

Очевидно, що обмін процесу із самим собою через канал великого смислу не має, тому зазвичай через канал взаємодіють два (або більше) процесів. Процес, що створює канал, прийнято називати сервером, а інший процес – клієнтом. Для спілкування з каналом клієнт і сервер повинні мати описувачі (дескриптори, handles) для читання і запису. Процес-сервер одержує описувач при створенні каналу. Процес-клієнт може одержати описувачі в результаті спадкоємства, у тому випадку, коли клієнт є нащадком

сервера. Це типово для спілкування через так звані анонімні канали. Інший спосіб отримання – відкриття на ім'я вже існуючого іменованого каналу неспорідненим процесом, який в результаті також стає володарем необхідних описувачів. Якщо організація доступу до каналу пройшла успішно, то схема взаємодії може виглядати так, як показано на рис. 5.8.

Рис. 5.8. Спілкування процесів через канал зв'язку

Якщо потрібно організувати однонаправлений зв'язок і ухвалено рішення про напрям передачі даних, то можна "закрити" невживаний кінець каналу. У прикладі на рис. 5.9 клієнт посилає через канал інформацію серверу.

Рис. 5.9. Передача інформації від клієнта сервера через канал зв'язку

Організація каналів в ОС Windows

Анонімні канали

Анонімні канали у Windows – це напівдуплексний засіб потокової передачі байтів між спорідненими процесами. Вони функціонують у межах локальної обчислювальної системи і добре підходять для перенаправлення вихідного потоку однієї програми на вхід іншої. Анонімні канали реалізовані за допомогою іменованих каналів з унікальними іменами. Анонімні канали створюються процесом-сервером за допомогою функції CreatePipe:

```
BOOL CreatePipe(
PHANDLE hReadPipe,                        // описувач для читання
PHANDLE hWritePipe,                       // описувач для запису
LPSECURITY_ATTRIBUTES lpPipeAttributes, // атрибути безпеки
DWORD nSize);                             // розмір каналу
```

Функція CreatePipe повертає два описувачі (дескриптори) для читання і запису в канал. Після створення каналу необхідно передати клієнтському процесу ці дескриптори (або один із них), що зазвичай робиться за допомогою механізму спадкоємства.

Для спадкоємства описувача потрібно, щоб дочірній процес створювався функцією CreateProcess із прапором спадкоємства TRUE. Заздалегідь потрібно створити успадковані описувачі. Це можна зробити, наприклад шляхом явної специфікації параметра bInheritHandle структури SECURITY_ATTRIBUTES при створенні каналу.

Іншим способом є створення успадкованого дубліката наявного описувача за допомогою функції DuplicateHandle і подальша передача його створюваному процесу через командний рядок або в який-небудь інший спосіб.

Одержавши потрібний описувач, клієнтський процес, так само як і сервер, може далі взаємодіяти з каналом за допомогою функцій ReadFile і WriteFile. Після закінчення роботи з каналом обидва процеси повинні закрити описувачі за допомогою функції CloseHandle.

Дослідження роботи програми спілкування процесу через анонімний канал із самим собою

У прикладі 5.3 наведено текст програми, під час роботи якої процес обмінюється інформацією із самим собою. У програмі створюється анонімний канал, у нього записується рядок цифр, потім частина цього рядка читається і виводиться на екран. Програмний додаток необхідно запускати в **консольному режимі**.

Приклад 5.3

```
uses windows;
var hRead, hWrite:HANDLE;
   BufIn,BufOut:array[0..100]of char;
   BufSize:dword=100;
   BytesOut:dword=10;
   BytesIn:dword=5;
   i:dword;
begin
   BufOut:='0123456789';
```

```
  if not CreatePipe(hRead,hWrite,NIL,BufSize) then
    writeln('Create pipe failed.');
  WriteFile(hWrite,BufOut,BytesOut,BytesOut,NIL);
   write('Write into pipe ',BytesOut,' bytes : ');
      for i:=0 to BytesOut do write(BufOut[i]);
  writeln;
  ReadFile(hRead,BufIn,BytesIn,BytesIn,NIL);
   write('Read from pipe ',BytesIn,' bytes : ');
   for i:=0 to BytesIn do write(BufIn[i]);
end.
```

Як самостійну вправу рекомендується організувати через анонімний канал взаємодію між окремими процесами (модель клієнт-сервер). Процес-сервер повинен створити канал, записати в нього інформацію і запустити процес-клієнт, завдання якого – витягнути записану інформацію з каналу.

Іменовані канали

Іменовані канали є об'єктами ядра ОС Windows, що дозволяють організувати міжпроцесний обмін не тільки в ізольованій обчислювальній системі, але і в локальній мережі. Вони забезпечують дуплексний зв'язок і дозволяють використовувати як потокову модель, так і модель, зорієнтовану на повідомлення. Обмін даними може бути синхронним і асинхронним.

Канали повинні мати унікальні в рамках мережі імена відповідно до правил іменування ресурсів у мережах Windows (Universal Naming Convention, UNC), наприклад \ServerName\pipe\PipeName. Для спілкування усередині одного комп'ютера ім'я записується у формі \.\pipe\PipeName, де "." позначає локальну машину. Слово "pipe" у складі імені фіксоване, а PipeName – ім'я, що задається користувачем. Ці імена, подібно до імен відкритих файлів, не є іменами об'єктів. Вони належать до простору імен під управлінням драйверів файлових систем іменованих каналів (\Winnt\System32\Drivers\Npfs.sys), прив'язаного до спеціального об'єкта пристрою \Device\NamedPipe, на яке є посилання в каталозі глобальних імен об'єктів \??\Pipe (ці останні імена "бачить" утиліта WinObj).

Імена створених іменованих каналів можна перерахувати за допомогою вільно поширюваної утиліти pipelist із сайта www.sysinternals.com. Оскільки імена каналів інтегровані в загальну структуру імен об'єктів, додатки можуть відкривати іменовані канали за допомогою функції CreateFile і взаємодіяти з ними через функції ReadFile і WriteFile.

Використання іменованих каналів

Сервер створює іменований канал за допомогою функції CreateNamedPipe (опис функції дивіться у відповідній літературі).

Окрім імені каналу, у вищеописаній формі до параметрів функції входять: прапор, який вказує модель передачі даних; параметр, що визначає синхронний або асинхронний режим роботи каналу, а також вказує, чи повинен канал бути одностороннім або двостороннім. Дотого ж існує необов'язковий дескриптор захисту, що забороняє несанкціонований доступ до іменованого каналу, і параметр, що визначає максимальну кількість одночасних з'єднань по даному каналу.

Повторно викликаючи CreateNamedPipe, можна створювати додаткові екземпляри цього ж каналу.

Після виклику CreateNamedPipe сервер виконує виклик ConnectNamedPipe і чекає відгуку від клієнтів, які з'єднуються з каналом за допомогою функції CreateFile або CallNamedPipe, указуючи при виклику ім'я створеного сервером каналу. Легальний клієнт одержує описувач, що представляє клієнтську сторону іменованого каналу, і робота серверної функції ConnectNamedPipe на цьому завершується.

Після того, як з'єднання по іменованому каналу встановлене, клієнт і сервер можуть використовувати його для читання і запису даних через Win32-функції ReadFile і WritePile.

Дослідження роботи програм спілкування двох процесів (клієнта і сервера) через іменований канал

Дослідимо роботу програм спілкування клієнта і сервера через іменований канал, тексти яких наведено у прикладах 5.4, 5.5. Програмні додатки необхідно запускати в **консольному режимі**.

Сервер:

Приклад 5.4

```
uses windows,crt;
var piProcInfo:tPROCESSINFORMATION;
   SI:tSTARTUPINFO ;
   hPipe:HANDLE;
   Buff:array[0..255]of char;
   iNumBytesToRead:Dword:=255;
   i:Dword;
begin
ZeroMemory(@SI,sizeof(tSTARTUPINFO));
SI.cb := sizeof(tSTARTUPINFO);
ZeroMemory(@piProcInfo, sizeof(piProcInfo));
  hPipe := CreateNamedPipe(
'\\.\pipe\MyPipe',                    // ім'я каналу
PIPE_ACCESS_DUPLEX,          // зчитування та запис із каналу
PIPE_TYPE_MESSAGE or        // передача повідомлень по каналу
PIPE_READMODE_MESSAGE or // режим зчитування повідомлень
PIPE_WAIT,                          // синхронна передача повідомлень
PIPE_UNLIMITED_INSTANCES,   // кількість екземплярів каналу
4096,                                  // розмір вихідного буфера
4096,                                   // розмір вхідного буфера
NMPWAIT_USE_DEFAULT_WAIT, // тайм-аут клієнта
NIL);                                   // захист за замовчуванням
  if hPipe=INVALID_HANDLE_VALUE then begin
     writeln('CreatePipe failed: error code ',GetLastError);end;
  if CreateProcess(NIL,'client.exe',NIL,NIL,FALSE,0,NIL,NIL,SI,
                                                  piProcInfo)=false then
     writeln('Create client process: error code ',GetLastError);
  if ConnectNamedPipe(hPipe,NIL)=false then
     writeln('Client could not connect');
  ReadFile(hPipe, Buff, iNumBytesToRead, iNumBytesToRead, NIL);
  TextBackground(15);textcolor(4);GotoXY(30,3);
  for i:=0 to iNumBytesToRead do write(Buff[i]);writeln;readln;
end.
```

Клієнт:

Приклад 5.5

```
uses windows;
var hPipe:HANDLE;
   i,NumBytesToWrite:dword;
   Buff:array[0..255]of char;
   tex:string:=‘ Message from Client !!!’;
begin
for i:=1 to length(tex)do Buff[i-1]:=tex[i];
  hPipe:=CreateFile(
       ‘\\.\pipe\MyPipe’,        // ім’я каналу
       GENERIC_READ or   // читання або запис у канал
       GENERIC_WRITE,
       0,                        // немає операцій, що розділяються
       NIL,                      // захист за замовчуванням
       OPEN_EXISTING,    // відкриття існуючого каналу
       0,                        // атрибути за замовчуванням
       0);                       // немає додаткових атрибутів
  WriteFile(hPipe, Buff, Length(tex), NumBytesToWrite, NIL);
end.
```

У даному прикладі процес-сервер створює канал, потім запускає процес-клієнт і чекає з’єднання. Далі він читає повідомлення, послане клієнтом.

Крім перерахованих вище, система представляє ще ряд корисних функцій для роботи з іменованими каналами. Для копіювання даних з іменованого каналу без видалення їх із каналу використовується функція PeekNamedPipe. Функція TransactNamedPipe застосовується для об’єднання операцій читання і запису в канал в одну операцію, яка називається транзакцією. Є інформаційні функції для визначення стану каналу, наприклад GetNamedPipeHandleState або GetNamedPipeInfo. Повний перелік знаходиться у відповідній довідниковій літературі.

Отже, організація спільної діяльності і спілкування процесів є важливим і актуальним завданням. До основних способів міжпроцесного обміну традиційно відносять канали і розподілену

пам’ять, для організації яких використовують розподілені ресурси. Анонімні канали підтримують потокову модель, у рамках якої дані є неструктурованою послідовністю байтів. Іменовані канали, що підтримують як потокову модель, так і модель, зорієнтовану на повідомлення, забезпечують обмін даними не тільки в ізольованому обчислювальному середовищі, але і в локальній мережі.

Практична частина

1. Складіть програми, наведені у прикладах 5.1 – 5.5 теоретичної частини даної лабораторної роботи. Запустіть програми та переконайтеся, що вони працюють правильно (згідно з описом алгоритму їх роботи).

2. Оформіть звіт із виконаної лабораторної роботи, який повинен містити текст програм, демонстрацію результатів роботи, та дайте відповіді на контрольні запитання.

Контрольні запитання та завдання

1. Які основні способи обміну інформацією між процесами ви знаєте?
2. Що собою являє розподілений ресурс?
3. Опишіть взаємодію клієнтів та сервера на прикладі роботи програм 5.1 та 5.2.
4. Опишіть особливості організації каналів зв’язку в операційних системах.
5. В чому різниця між анонімними та іменованими каналами?

Лабораторна робота № 6
Дослідження алгоритмів та механізмів синхронізації процесів у операційній системі Windows. Моделювання паралельних систем та усунення тупикових ситуацій

Теоретична частина

Вступ. Проблема взаємовиключення

Взаємозв'язані потоки, які обмінюються даними або користуються одними і тими ж пристроями введення-виведення, повинні синхронізувати свою роботу. Знехтування питаннями синхронізації потоків, що виконуються в режимі мультипрограмування, може призвести до їх неправильної роботи або навіть до краху системи. Проблема синхронізації, яка виникає в подібних випадках, може розв'язуватися припиненням і активізацією потоків, організацією черг, блокуванням і звільненням ресурсів.

Припустимо, що два потоки, що фіксують які-небудь події, намагаються дати приріст загальної змінної Count, лічильнику цих подій (рис 6.1).

Counter: integer;

$Thread_1$	$Thread_2$
…	…
Inc(Count);	
	Inc(Count);

Рис. 6.1. Два паралельні потоки збільшують значення спільної змінної Count

Операція Inc(Count) не є атомарною. Код операції Inc(Count) буде перетворений компілятором у машинний код, який виглядає приблизно так:

```
(1) MOV EAX [Count]  ;значення з Count поміщається в регістр
(2) INC EAX          ;значення регістра збільшується на 1
(3) MOV [Count], EAX ;значення з регістра поміщається назад у Count
```

У мультипрограмній системі з розділенням часу може виникнути несприятлива ситуація перемішування (interleaving), коли потік T_1 виконує крок (1), потім витісняється потоком T_2, який виконує кроки (1) – (3), а вже після цього потік T_1 закінчує операцію, виконуючи кроки (2) – (3). В цьому випадку результуючий приріст змінної Count дорівнюватиме 1 замість правильного приросту – 2.

Складність проблеми синхронізації полягає в нерегулярності виникаючих ситуацій: у попередньому прикладі можна представити й інший, сприятливіший розвиток подій. У даному випадку все визначається взаємними швидкостями потоків і моментами їх переривання. Ситуації, подібні тій, коли два або більш потоків обробляють дані, що розділяються, і кінцевий результат залежить від співвідношення швидкостей процесів, називаються *гонками (умови змагання, race conditions)*.

Для усунення умов змагання необхідно забезпечити кожному потоку ексклюзивний доступ до даних, що розділяються. Такий прийом називається взаємовиключенням (mutual exclusion). Частина коду потоку, виконання якого може привести до race condition, називається *критичною секцією (critical section)*. Наприклад, операції (1) – (3) в прикладі, наведеному вище, є критичними секціями обох потоків. Отже, взаємовиключення необхідно забезпечити для критичних секцій потоків.

У загальному випадку структура процесу, що бере участь у взаємодії, може бути представлена так:

```
while (some condition) do begin
entry section
 critical section
 exit section
 remainder section          end;
```

Зовнішній цикл означає, що нас цікавитимуть численні спроби входу в критичну секцію (синхронізація одиничних попадань може бути забезпечена й іншими засобами). Найбільш важливий із погляду синхронізації пролог (entry section), де ухвалюється рішення про те, чи може потік бути допущеним у критичну секцію. У епілогу (exit section) зазвичай відкривається шлагбаум

для інших потоків, а операції, що не входять у критичну секцію, зосереджені в remainder section.

Змінна-замок

Одним із можливих не цілком коректних розв'язків проблеми синхронізації є використання змінної-замка. Наприклад, можна зробити умовою входження в критичну секцію значення 0 деякої змінної lock, що розділяється. Відразу ж після перевірки це значення змінюється на 1 (закриття замка). При виході з критичної секції замок відкривається (значення змінної lock скидається в 0).

var Lock: integer=0; // shared-змінна, тобто змінна, спільна для обох потоків T_1 та T_2

```
 T1                                T2
      while (some condition) do begin
      while(lock) do begin  end;
       lock:= 1;
            critical section
       lock:= 0;
            remainder section      end;
```

На жаль, запропонований розв'язок не завжди забезпечує взаємовиключення. Унаслідок того, що дія-пролог, що складається з двох операцій while(lock); lock:= 1; не є атомарною, існує відмінна від нуля імовірність витіснення потоку між цими операціями. При цьому управління може перейти до другого потоку, який, дізнавшись, що змінна lock все ще дорівнює 0, може увійти до своєї критичної секції.

Отже, проблема синхронізації може бути розв'язана за рахунок забезпечення безперервності для декількох операцій, серед яких є операції опитування поточного значення деякої змінної і встановлення для цієї змінної нового значення.

TSL команди

Багато обчислювальних систем мають інструкції, які можуть забезпечити атомарність послідовності операцій при вході в критичну секцію і підтримуються на апаратному рівні (на рівні команд мікропроцесорів). Такі команди називаються Test and_Set Lock (перевірити та встановити 1) або TSL командами. Якщо уявити собі таку команду як функцію:

```
function TSL (var target:integer):integer;
begin
TSL:= target;
target: = 1;
end;
```

то, замінивши в попередньому прикладі послідовність операцій while(lock); lock: = 1; на TSL(lock), ми одержуємо розв'язок проблеми взаємовиключення.

Сімейство Interlocked-функцій

Сімейство Interlocked-функцій, що входить до складу Win32 API і виконуються атомарно, дає ключ до рішення багатьох питань синхронізації. Наприклад, функція

LONG InterlockedExchangeAdd(PLONG plAddend, LONG lncrement);

дозволяє атомарним чином збільшити значення змінної. В цьому випадку коректний приріст змінної Count, описаний на початку лабораторної роботи, може бути забезпечений такою операцією:

InterlockedExchangeAdd (@Count, 1);

У відповідній літературі можна прочитати і про інші Interlocked-функції. Наприклад, як TSL-інструкції, необхідної для розв'язання проблеми входу в критичну секцію, можна застосувати функцію InterlockedCompareExchange.

Реалізація Interlocked-функцій залежить від апаратної платформи. На x86-процесорах вони видають на шину апаратний сигнал, закриваючи для інших процесорів конкретну адресу пам'яті.

Істотно те, що Interlocked-функції виконуються в користувацькому режимі роботи процесора протягом приблизно 50 тактів, тобто надзвичайно швидко.

Дослідження роботи програми синхронізації за допомогою змінної-замка

Для практичного знайомства з проблемою синхронізації спочатку необхідно написати програму, що вимагає синхронізації. Наприклад, таку, як наведена нижче програма async (***Запускати слід у консольному режимі***):

Приклад 6.1

```
uses windows;
var Sum:integer=0;iNumber:integer=5;jNumber:integer=30000000;
procedure SecondThread;
var i,j:integer;
begin
  for i:=1 to iNumber do begin Sum:=0;
  for j:=1 to jNumber do Sum:=Sum+1;
  writeln('        ',Sum); end; end;
var i,j:integer;
    hThread:HANDLE;
    IDThread:DWORD;
begin
  hThread:=CreateThread(NIL,0,@SecondThread,NIL,0,IDThread);
  if @hThread=NIL then exit;
  for i:=1 to iNumber do begin Sum:=0;
  for j:=1 to jNumber do Sum:=Sum-1;
  writeln(Sum);          end;
  WaitForSingleObject(hThread, INFINITE); end.
```

У даній програмі потік SecondThread у циклі дає приріст загальній змінній Sum, а основний потік також у циклі зменшує її значення і періодично виводить його на екран. Легко переконатися, що результати роботи програми внаслідок перемішування непередбачувані, особливо якщо параметр jNumber підібрати з урахуванням швидкодії комп'ютера.

Рекомендується ввести в дану програму синхронізацію за допомогою глобальної змінної-замка, включивши в неї операції while(lock); і lock:=1; і добитися передбаченості в роботі програми.

Оскільки ситуація, в якій виділений потоку квант часу закінчується між while(lock); і lock:=1; малоймовірна, можна змоделювати її штучно, ввівши між цими операціями паузу (функція Sleep, наприклад). Нарешті, бажано реалізувати правильне рішення шляхом опиту і модифікації змінної-замка за допомогою TSL-інструкції (функція InterlockedCompareExchange).

Спін-блокування

Розглянуті розв'язання проблеми синхронізації, безумовно, коректні. Вони реалізують такий алгоритм: перед входом у критичну секцію потік перевіряє можливість входу і, якщо такої можливості немає, продовжує опитування значення змінної-замка. Така поведінка потоку, пов'язана з його обертанням в порожньому циклі, називається активним очікуванням або блокуванням спіну (spin lock).

Очевидно, що на однопроцесорній машині це марна трата машинного часу, оскільки значення опитуваної змінної протягом цього циклу не може бути жодним чином змінено.

Спін-блокування, хоча б тимчасове, може бути корисним на багатопроцесорних машинах, де один із потоків крутиться в циклі, а другий – працює на іншому процесорі і може змінити значення змінної-замка. В цьому випадку в активно очікуючого потоку є шанс швидко увійти до критичної секції, не будучи блокованим. Блокування потоку пов'язане з переходом у режим ядра (приблизно 1000 тактів роботи процесора).

Проте розумнішим рішенням, особливо на однопроцесорних машинах, видається переведення потоку в стан очікування, якщо вхід у критичну секцію заборонений. Після зняття заборони на вхід у критичну секцію цей потік переводиться в стан готовності.

Critical Sections

У складі API ОС Windows є спеціальні і ефективні функції для організації входу в критичну секцію і виходу з неї потоків одного процесу в режимі користувача. Вони називаються EnterCriticalSection і LeaveCriticalSection і мають як параметр структуру попередньо ініційованого типу CRITICAL_SECTION.

Зразкова схема програми може виглядати так:

```
var cs: CRITICAL_SECTION;
procedure SecondThread;
begin
 InitializeCriticalSection(@cs);
EnterCriticalSection(@cs);
// критична ділянка коду
LeaveCriticalSection(@cs); end;
```

```
begin
InitializeCriticalSection(@cs);
CreateThread(NIL, 0, @SecondThread, …);
EnterCriticalSection(@cs);
// критична ділянка коду
LeaveCriticalSection(@cs);
DeleteCriticalSection(@cs);
end.
```

Функції EnterCriticalSection і LeaveCriticalSection реалізовані на основі Interlocked-функцій, виконуються атомарним чином і працюють дуже швидко. Істотно те, що у випадку неможливості входу в критичну ділянку потік переходить у стан очікування. Згодом, коли така можливість з'явиться, потік буде "пробуджений" і зможе зробити спробу входу в критичну секцію. Механізм пробудження потоку реалізований за допомогою об'єкта ядра "подія" (event), яка створюється тільки у випадку виникнення конфліктної ситуації.

Вже говорилося, що іноді, перед блокуванням потоку, є зміст якийсь час утримувати його в стані активного очікування. Щоб функція EnterCriticalSection виконувала задану кількість циклів спін-блокування, критичну секцію доцільно ініціювати за допомогою функції InitalizeCriticalSectionAndSpinCount.

Як самостійну вправу рекомендується реалізувати синхронізацію у вищенаведеній програмі async (приклад 6.1) за допомогою перерахованих примітивів. Важливо не забувати про коректний вихід із критичної секції, тобто про парне використання функцій EnterCriticalSection і LeaveCriticalSection.

Синхронізація потоків із використанням об'єктів ядра

Критичні секції, розглянуті в попередньому розділі, підходять для синхронізації потоків одного процесу. Завдання синхронізації потоків різних процесів прийнято виконувати за допомогою об'єктів ядра. Об'єкту ядра може бути привласнено ім'я, вони дозволяють задавати тайм-аут для часу очікування і володіють ще рядом можливостей для реалізації гнучких сценаріїв синхронізації. Проте їх використання пов'язане з переходом у режим ядра

(приблизно 1000 тактів процесора), тобто вони працюють дещо повільніше, ніж критичні секції.

Майже всі об'єкти ядра, розглянуті раніше, зокрема процеси, потоки і файли, придатні для виконання завдань синхронізації. У контексті завдань синхронізації про кожний з об'єктів можна сказати, чи знаходиться він у вільному (сигнальному, signaled state) або зайнятому (nonsignaled state) стані. Правила переходу об'єкта з одного стану в інший залежать від самого об'єкта. Наприклад, якщо потік виконується, то він знаходиться в зайнятому стані, а якщо потік успішно завершив очікування семафора, то семафор знаходиться в зайнятому стані.

Потоки знаходяться в стані очікування, поки очікувані ними об'єкти зайняті. Як тільки об'єкт звільняється, ОС будить потік і дозволяє продовжити виконання. Для припинення потоку і переведення його в стан очікування звільнення об'єкта використовується функція

DWORD WaitForSingleObject(HANDLE hObject, DWORD dwMilliseconds);

де hObject – описувач очікуваного об'єкта ядра, а другий параметр – максимальний час очікування об'єкта.

Потік створює об'єкт ядра за допомогою сімейства функцій Create (CreateSemaphore, CreateThread і т.д.), після чого об'єкт за допомогою описувача стає доступним усім потокам даного процесу. Копія описувача може бути одержана за допомогою функції DuplicateHandle і передана іншому процесу, після чого потоки зможуть скористатися цим об'єктом для синхронізації.

Іншим, поширенішим способом отримання описувача є відкриття існуючого об'єкта по імені, оскільки багато об'єктів мають імена в просторі імен об'єктів. Ім'я об'єкта – один із параметрів *Create-функцій.* Знаючи ім'я об'єкта, потік, що володіє потрібними правами доступу, одержує його описувач за допомогою Open-функцій. Нагадаємо, що в структурі, що описує об'єкт, є лічильник посилань на нього, який збільшується на 1 при відкритті об'єкта і зменшується на 1 при його закритті.

Дещо докладніше розглянемо ті об'єкти ядра, які призначені безпосередньо для розв'язання проблем синхронізації.

Семафори

Відомо, що семафори, запропоновані Дейкстрой в 1965 р., є цілою змінною в просторі ядра, доступ до якої, після її ініціалізації, може здійснюватися через дві атомарні операції: wait і signal (у ОС Windows це функції WaitForSingleObject і ReleaseSemaphore відповідно).

wait(S): якщо S <= 0 процес блокується
(переводиться в стан очікування);
інакше S = S – 1;

signal(S): S = S + 1

Семафори зазвичай використовуються для обліку ресурсів (поточна кількість ресурсів задається змінною S) і створюються за допомогою функції CreateSemaphore, до параметрів якої входять початкове і максимальне значення змінної. Поточне значення не може бути більше від максимального і негативним. Значення S, яке дорівнює нулю, означає, що семафор зайнятий.

Нижче наведений приклад синхронізації програми async за допомогою семафорів.

Приклад 6.2

```
uses windows;
var Sum:integer=0;iNumber:integer=5;jNumber:integer=30000000;
   hFirstSemaphore,hSecondSemaphore:HANDLE;
procedure SecondThread;
var i,j:integer;
begin
  for i:=1 to iNumber do begin
   WaitForSingleObject(hSecondSemaphore,INFINITE);
   Sum:=0;
   for j:=1 to jNumber do Sum:=Sum+1;
 writeln('       ',Sum);
 ReleaseSemaphore(hFirstSemaphore, 1, NIL); end;
end;
var i,j:integer;
   hThread:HANDLE;
   IDThread:DWORD;
begin
```

```
hFirstSemaphore:=CreateSemaphore(NIL,0,1,'MyFirstSemaphore');
hSecondSemaphore:=CreateSemaphore(NIL,1,1,
                                              'MySecondSemaphore');
  hThread:=CreateThread(NIL,0,@SecondThread,NIL,0,IDThread);
  if @hThread=NIL then exit;
  for i:=1 to iNumber do begin
  WaitForSingleObject(hFirstSemaphore, INFINITE);
  Sum:=0;
  for j:=1 to jNumber do Sum:=Sum-1;
  writeln(Sum);
  ReleaseSemaphore(hSecondSemaphore,1,NIL); end;
WaitForSingleObject(hThread,INFINITE);
CloseHandle(hFirstSemaphore);
CloseHandle(hSecondSemaphore);
end.
```

У даній програмі синхронізація дій двох потоків, що забезпечує однаковий результат для всіх запусків програми, виконана за допомогою двох семафорів, приблизно так, як це робиться в завданні producer-consumer (виробник-споживач). Потоки по черзі відкривають один одному дорогу до критичної ділянки. Першим починає працювати потік SecondThread, оскільки значення лічильника утримуючого його семафора ініційовано одиницею при створенні цього семафора. Синхронізацію за допомогою семафорів потоків різних процесів рекомендується виконати як самостійну вправу.

М'ютекси

М'ютекси також є об'єктами ядра, використовуваними для синхронізації, але вони простіше за семафори, оскільки регулюють доступ до єдиного ресурсу, а отже, не містять лічильників. По суті вони поводяться як критичні секції, але можуть синхронізувати доступ потоків різних процесів. Ініціалізація м'ютекса здійснюється функцією CreateMutex, для входу в критичну секцію використовується функція WaitForSingleObject, а для виходу – ReleaseMutex.

Якщо потік завершується, не звільнивши м'ютекс, останній переходить у вільний стан. Відмінність від семафорів у тому, що

потік, який зайняв м'ютекс, отримує права на володіння ним. Тільки цей потік може звільнити м'ютекс. Тому думка про м'ютекс як про семафор із максимальним значенням 1 не цілком відповідає дійсності.

Події

Об'єкти "події" – найбільш примітивні об'єкти ядра. Вони призначені для інформування одного потоку іншим про закінчення якої-небудь операції. Події створюються функцією CreateEvent. Простий варіант синхронізації: переводити подію в зайнятий стан функцією WaitForSingleObject і у вільний – функцією SetEvent.

У різноманітних керівництвах із програмування розглядаються більш складні сценарії, пов'язані з типом події (які скидаються вручну і які скидаються автоматично) і з управлінням синхронізацією груп потоків, а також ряд додаткових корисних функцій.

Розробку програм, в яких для виконання завдань синхронізації використовуються м'ютекси і події, рекомендується виконати як самостійну вправу.

Сумарні відомості про об'єкти ядра

У різноманітних керівництвах із системного програмування містяться відомості і про інші об'єкти ядра стосовно синхронізації потоків.

Зокрема, існують такі властивості об'єктів:

- процес і потік знаходяться в зайнятому стані, коли активні, і у вільному стані, коли завершуються;
- файл знаходиться в зайнятому стані, коли виданий запит на уведення-виведення, і у вільному стані, коли операція введення-виведення завершена;
- повідомлення про зміну файла знаходиться в зайнятому стані, коли у файловій системі немає змін, і у вільному – коли зміни виявлені;
- і т.д.

Синхронізація в ядрі

Розв'язання проблеми взаємовиключення особливо актуальне для такої складної системи, як ядро ОС Windows.

Одна з проблем пов'язана з тим, що код ядра часто працює на пріоритетних IRQL рівнях "DPC/dispatch" або "вище", відомих як

"високий IRQL". Це означає, що традиційні засоби синхронізації, пов'язані з припиненням потоку, не можуть бути використані, оскільки процедура планування і запуску іншого потоку має нижчий пріоритет. Водночас існує небезпека виникнення події, чий IRQL вище, ніж IRQL критичної ділянки, яка буде в цьому випадку витіснена. Тому в подібних ситуаціях вдаються до прийому, який називається "заборона переривань". У випадку Windows цього добиваються, штучно підвищуючи IRQL критичної ділянки до найвищого рівня, використовуваного будь-яким можливим джерелом переривань. У результаті критична ділянка може безперешкодно виконати свою роботу.

На жаль, для мультипроцесорних систем подібна стратегія не годиться. Заборона переривань на одному з процесорів не виключає переривань на іншому процесорі, який може продовжити свою роботу і дістати доступ до критичних даних. У цьому випадку потрібен спеціальний протокол установки взаємовиключення. Основою цього протоколу є установка блокуючої змінної (змінної-замка), зіставленої з кожною глобальною структурою даних, за допомогою TSL команди. Оскільки установка замка відбувається в результаті активного очікування, то кажуть, що код ядра встановлює (захоплює) спін-блокування. Установка спін-блокування відбувається при високих IRQL рівнях, тому код ядра, який захоплює спін-блокування та утримує його для виконання критичної секції коду, ніколи не витісняється. Встановлення та звільнення спін-блокувань здійснюється функціями ядра KeAcquireSpinlock і KeReleaseSpinlock, які активно використовуються в ядрі та драйверах пристроїв. На однопроцесорних системах установка і зняття спін-блокувань реалізується простим підвищенням і пониженням IRQL.

Нарешті, маючи набір глобальних ресурсів, в даному випадку – спін-блокувань, необхідно розв'язати проблему виникнення потенційної безвиході (тупикової ситуації). Наприклад, потік 1 захоплює блокування 1, а потік 2 – блокування 2. Потім потік 1 намагається захопити блокування 2, а потік 2 – блокування 1. В результаті обидва потоки ядра виснуть. Одним із розв'язків даної проблеми є нумерація всіх ресурсів і виділення їх тільки у поряд-

ку зростання номерів. У випадку Windows є ієрархія спінблокувань: всі вони поміщаються в список у порядку убування частоти використання і повинні захоплюватися в тому порядку, в якому вказані в списку.

У випадку низьких IRQL синхронізація здійснюється традиційно – за допомогою об'єктів ядра.

Тупикові ситуації

До цього моменту ми розглядали способи синхронізації процесів, які дозволяють процесам успішно кооперуватися. Проте в деяких випадках можуть виникнути непередбачені ускладнення. Припустимо, що декілька процесів конкурують за володіння кінцевою кількістю *ресурсів*. Якщо запрошуваний процесом *ресурс* недоступний, ОС переводить даний процес у стан очікування. У разі коли необхідний *ресурс* утримується іншим очікуючим процесом, перший процес не зможе змінити свій стан. Така ситуація називається ***безвихіддю (deadlock)***. Говорять, що в мультипрограмній системі процес знаходиться в стані *безвиході*, якщо він чекає події, яка ніколи не відбудеться. *Ресурсами* можуть бути як пристрої, так і дані. Деякі *ресурси* допускають розділення між процесами, тобто є *ресурсами, що* **розділяються**. Системна *тупикова ситуація*, або "зависання системи", є наслідком того, що один або більше процесів знаходяться в стані *безвиході*. Іноді подібні ситуації називають ***взаємоблокуваннями***. У загальному випадку проблема *безвиході* ефективного розв'язку не має. Множина процесів знаходиться в *тупиковій ситуації*, якщо кожен процес із множини чекає події, яка може викликати тільки інший процес даної множини. Оскільки всі процеси чогось чекають, то жоден із них не зможе ініціювати подію, яка „розбудила" б іншого члена множини, а отже, всі процеси „спатимуть" разом.

Розробка операційних систем охоплює множину цікавих проблем, які широко обговорювались та аналізувались з використанням різних методів синхронізації. Розглянемо найбільш відому проблему „філософів".

У 1965 році Дейкстра сформулював та розв'язав проблему синхронізації, названу ним *проблемою філософів, які обідають*. Відтоді кожен, хто винаходив черговий примітив синхронізації,

вважав своїм обов'язком продемонструвати переваги нового примітиву на прикладі проблеми філософів, які обідають. Задачу можна сформулювати так: п'ять філософів сидять за круглим столом і в кожного є тарілка зі спагеті. Сагеті настільки слизькі, що кожному філософу необхідно дві виделки, щоб їсти. Але між кожними двома тарілками лежить одна виделка (рис. 6.2).

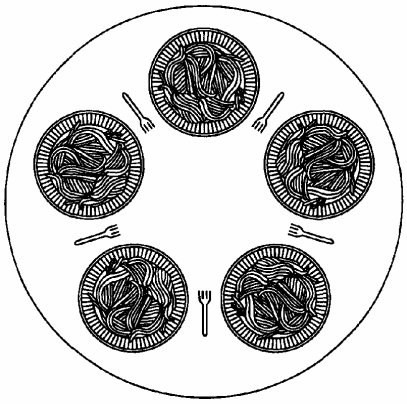


Рис. 6.2. Проблема філософів, які обідають

Життя філософа складається з почергових періодів споживання їжі та роздумів. Коли філософ голодний, він намагається отримати дві виделки, ліву і праву, в будь-якому порядку. Якщо йому вдалося заволодіти двома виделками, він деякий час їсть, потім кладе виделки назад і продовжує роздуми. Питання полягає ось у чому: чи можна створити алгоритм, який моделює ці дії для кожного філософа і ніколи не застрягає? Очевидно, що проста модельна реалізація дій філософів може призвести до ситуації, коли філософи приблизно однаково в часі зголодніють, почнуть обирати виделки, і в кожного може опинитися по одній лівій або правій виделці, тобто виникне ситуація взаємоблокування і безвиході.

Для запобігання тупиковій ситуації необхідно передбачити певну синхронізацію для організації взаємовиключення. Дейкстра запропонував підхід семафорів. Якщо взяти бінарний семафор, то тупикової ситуації можна уникнути, але натомість, у кожен конкретний момент часу, лише один філософ може споживати їжу

(лише один процес може виконуватися). Якщо використати масив семафорів (по одному на кожного філософа), то можна підвищити ефективність паралельного виконання невзаємодіючих із точки зору спільних ресурсів процесів.

У прикладі 6.3 наведено текс програми, яка являє собою модель філософів. На рис. 6.3 показано результати її роботи у випадку виникнення тупикової ситуації, та при нормальній роботі. Модель передбачає наявність п'ятьох станів філософів, які відображаються різними кольорами: 1) стан роздумів (зелений); 2) стан голоду (червоний); 3) наявність однієї виделки (світло сірий); 4) наявність двох виделок (темно сірий); 5) споживання їжі (синій). У кожному з цих станів філософ може перебувати протягом деякого випадкового проміжку часу, по завершенні якого філософ готовий перейти у наступний стан. Синхронізація в даному випадку організована за допомогою масиву змінних-замків Lk. А для визначення атомарних критичних секцій використано API-функції EnterCriticalSection і LeaveCriticalSection. Якщо у функціях потоків A, B, C, D, E (моделі філософів) прибрати рядки repeat until((Lk[tL]=0)and(Lk[tR]=0)); (перевірка наявності вільних ресурсів – виделок), то отримуємо тупикову ситуацію, коли в кожного філософа по одній виделці і вони всі в очікуванні другої (рис. 6.3 зліва).

Приклад 6.3

```
uses Windows,Messages,math;
const N=5;
type ThreadManager =  packed record
    hwndParent: HWND;
    st_f:integer;
    st_L:integer;
    st_R:integer;                  end;
var DC: hDC;
   ps:TPaintStruct;
   Rgn: HRgn;
   hThreads:array[1..N]of HANDLE;
   Tm:array[1..N]of ThreadManager;
```

```
   Lk,ss,y,x,ys,xs:array[1..N]of integer;
   p:array[1..N]of pointer;
   cs:trtlCriticalSection;
   i,j,r,g,b:byte;
procedure T_S_F(k,s:byte);
begin
Tm[k].st_f:=s;InvalidateRect(Tm[k].hwndParent,nil,true);
sleep(1000+random(1000)); end;
procedure ThreadFuncA;
var t,tL,tR:byte;
begin t:=1;
if t=1 then tL:=5 else tL:=t-1;tR:=1+(t mod 5);
while(true)do begin T_S_F(t,0);T_S_F(t,1);
repeat until((Lk[tL]=0)and(Lk[tR]=0));
EnterCriticalSection(cs);
Lk[t]:=1;
repeat until((Tm[t].st_L=0)or(Tm[t].st_R=0));
if Tm[t].st_L=0 then begin
Tm[t].st_L:=1;Tm[tL].st_R:=1;ss[tL]:=1;end else
                 begin Tm[t].st_R:=1;Tm[tR].st_L:=1;ss[t]:=1;end;
LeaveCriticalSection(cs);
                T_S_F(t,3);
EnterCriticalSection(cs);
repeat until ((Tm[t].st_L=0)or(Tm[t].st_R=0));
if Tm[t].st_L=0 then begin
Tm[t].st_L:=1;Tm[tL].st_R:=1;ss[tL]:=1;end else
                 begin Tm[t].st_R:=1;Tm[tR].st_L:=1;ss[t]:=1;end;
LeaveCriticalSection(cs);
                T_S_F(t,4);T_S_F(t,2);
EnterCriticalSection(cs);
Tm[t].st_R:=0;Tm[tR].st_L:=0;ss[t]:=0;Tm[t].st_L:=0;
Tm[tL].st_R:=0;ss[tL]:=0;Lk[t]:=0;
LeaveCriticalSection(cs);
end;end;
```

.....//Чотири процедури – аналоги ThreadFuncA:
// ThreadFuncB (t:=2;) ThreadFuncC (t:=3;) ThreadFuncD (t:=4;) ThreadFuncE (t:=5;)

```
function WindowProc conv arg_stdcall
(Window:HWND;Mess:UINT;Wp:WParam;Lp:LParam):LRESULT;
begin
  case Mess of
      WM_CREATE:
      begin
      for j:=1 to N do begin
      Tm[j].hwndParent:=Window;
      hThreads[j]:=CreateThread(NIL,0,p[j],@Tm[j],0,hInstance);end;
      end;
      WM_PAINT:
      begin
      DC:= BeginPaint(Window, ps);
      for i:=1 to N do begin
      case Tm[i].st_f of
      0:begin r:=0;g:=255;b:=0;end;
      1:begin r:=255;g:=0;b:=0;end;
      2:begin r:=0;g:=0;b:=255;end;
      3:begin r:=200;g:=200;b:=200;end;
      4:begin r:=150;g:=150;b:=150;end;
      end;
      Rgn:=CreateEllipticRgn(x[i]-20,y[i]-20,x[i]+20,y[i]+20);
      FillRgn(DC,Rgn,CreateSolidBrush(RGB(r,g,b)));
      if Tm[i].st_f=3 then begin
      Rgn:=CreateEllipticRgn(x[i]-5,y[i]-5,x[i]+5,y[i]+5);
      FillRgn(DC,Rgn,CreateSolidBrush(RGB(0,0,0)));end;
      if(Tm[i].st_f=4)or(Tm[i].st_f=2)then begin
      Rgn:=CreateEllipticRgn(x[i]-5-5,y[i]-5,x[i]+5-5,y[i]+5);
      FillRgn(DC,Rgn,CreateSolidBrush(RGB(0,0,0)));
      Rgn:=CreateEllipticRgn(x[i]-5+5,y[i]-5,x[i]+5+5,y[i]+5);
      FillRgn(DC,Rgn,CreateSolidBrush(RGB(0,0,0)));end;
      if ss[i]=0 then begin
      Rgn:=CreateEllipticRgn(xs[i]-5,ys[i]-5,xs[i]+5,ys[i]+5);
      FillRgn(DC,Rgn,CreateSolidBrush(RGB(0,0,0)));end;end;
      EndPaint(Window, ps);
      end;
```

```
      WM_DESTROY:
      begin
      for j:=1 to N do CloseHandle(hThreads[j]);
      DeleteCriticalSection(cs);
      PostQuitMessage(0);       end;
  else Result := DefWindowProc(Window, Mess, Wp, Lp);
  end;
end;
var wc: TWndClass;wnd:HWnd;Msg: TMsg;Menu: hMenu;
    rad,a:extended;
begin randomize;InitializeCriticalSection(cs);
rad:=150;a:=-pi/2;for i:=1 to N do begin
x[i]:=250+round(rad*cos(a));y[i]:=250+round(rad*sin(a));
                                                    a:=a+2*pi/5; end;
rad:=100;a:=pi/2;for i:=1 to 3 do a:=a+2*pi/5; for i:=1 to N do begin
xs[i]:=250+round(rad*cos(a));ys[i]:=250+round(rad*sin(a));
                                                    a:=a+2*pi/5; end;
p[1]:=@ThreadFuncA;p[2]:=@ThreadFuncB;p[3]:=@ThreadFuncC;
p[4]:=@ThreadFuncD;p[5]:=@ThreadFuncE;
  FillChar(wc, SizeOf(wc), 0);
  with wc do begin
    style:=CS_HREDRAW + CS_VREDRAW;
    lpfnWndProc := @WindowProc;
    cbClsExtra := 0;
    cbWndExtra := 0;
    hInstance := System.hInstance;
    hIcon := LoadIcon(THandle(NIL), IDI_APPLICATION);
    hCursor := LoadCursor(THandle(NIL), IDC_ARROW);
    hbrBackGround := GetStockObject(WHITE_BRUSH);
    lpszMenuName := nil;
    lpszClassName := 'Лабораторна робота № 6';
          end;
  if RegisterClass(wc) = 0 then Exit;
    wnd := CreateWindow(wc.lpszClassName,'Програма дослідження
задачі філософів. Безвихідь усунено',
WS_OVERLAPPEDWINDOW,200,200,500,500,0,0,HInstance,nil);
```

```
ShowWindow(wnd, SW_RESTORE);
UpdateWindow(wnd);
while GetMessage(Msg,0,0,0) do
begin
  TranslateMessage(Msg);
  DispatchMessage(Msg);  end; end.
```

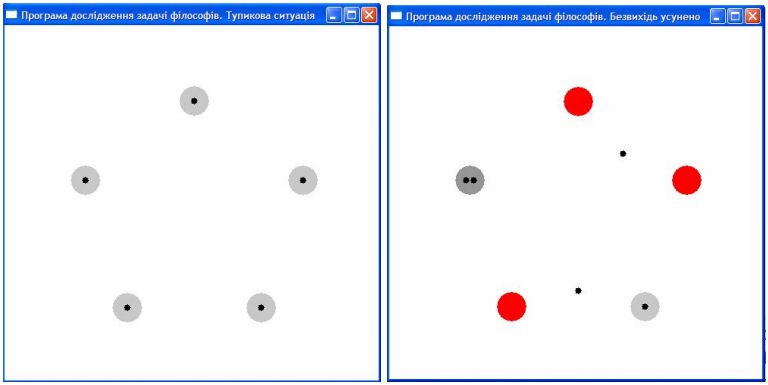


Рис. 6.3. Тупикова ситуація в задачі філософів (зліва);
нормальна робота паралельних процесів (справа)

Висновки

Отже, проблема недетермінізму є однією з ключових у паралельних обчислювальних середовищах. Традиційне рішення – організація взаємовиключень. Для синхронізації із застосуванням змінної-замка використовуються Interlocked-функції, що підтримують атомарність деякої послідовності операцій. Взаємовиключення потоків одного процесу найлегше організувати за допомогою примітиву Crytical Section. Для складніших сценаріїв рекомендується застосовувати об’єкти ядра, зокрема семафори, м’ютекси і події. У даній лабораторній роботі також розглянута проблема синхронізації в ядрі, основним розв’язком якої можна вважати встановлення і звільнення спін-блокувань, а також проблема тупикових ситуацій при моделюванні паралельних динамічних систем.

Практична частина

1. Складіть програми, наведені у прикладах 6.1 – 6.3 теоретичної частини даної лабораторної роботи. Запустіть програми та переконайтеся, що вони працюють правильно (згідно з описом алгоритму їх роботи).

2. На основі програми прикладу 6.1 дослідіть особливості використання різноманітних відомих вам алгоритмів синхронізації та об'єктів синхронізації ОС.

3. Оформіть звіт із виконаної лабораторної роботи, який повинен містити текст програм, демонстрацію результатів роботи, та дайте відповіді на контрольні запитання.

Контрольні запитання та завдання

1. Опишіть проблему взаємовиключення.
2. Які алгоритми синхронізації ви знаєте?
3. Які методи синхронізації ви знаєте?
4. Що таке тупикові ситуації (ситуації безвиході) та які умови їх виникнення?
5. Як можна уникнути тупикових ситуацій?
6. Опишіть особливості синхронізації в ядрі ОС Windows.
7. Розкрийте суть таких понять: активність, атомарна операція, критична секція, умови змагань, пролог, епілог.

Лабораторна робота № 7
Дослідження системи керування пам'яттю операційної системи Windows

Теоретична частина

Вступ

У комп'ютерах фон-нейманівської архітектури виконувані програми разом з оброблюваними ними даними повинні знаходитися в оперативній пам'яті. Операційній системі доводиться займатися управлінням пам'яттю, тобто розв'язувати задачу розподілу пам'яті між процесами користувача і компонентами ОС. Частина ОС, яка відповідає за управління пам'яттю, називається *менеджером пам'яті*.

Для опису системи управління пам'яттю активно використовуються поняття фізичної і логічної (віртуальної) пам'яті.

Фізична пам'ять є апаратним запам'ятовуючим пристроєм комп'ютера. Менеджер пам'яті має справу з двома рівнями фізичної пам'яті: оперативною (основною, первинною) і зовнішньою, або вторинною. Оперативна пам'ять виготовляється із застосуванням напівпровідникових технологій і втрачає свій вміст при відключенні живлення. Вторинна пам'ять (це, головним чином, диски) характеризується набагато повільнішим доступом, проте має велику місткість і є незалежною. Вона використовується як розширення основної пам'яті. Зазвичай інформація, що зберігається в оперативній пам'яті, за винятком самих останніх змін, зберігається також у зовнішній пам'яті. Якщо процесор не виявляє потрібну інформацію в оперативній пам'яті, він починає шукати її у вторинній. Коли потрібна інформація знайдена в зовнішній пам'яті, вона переноситься в оперативну пам'ять. Менеджер пам'яті прагне по можливості понизити частоту звернень до вторинної пам'яті (властивість локальності або локалізації звернень). У результаті ефективний час доступу до пам'яті виявляється близьким до часу доступу до оперативної пам'яті і складає декілька десятків наносекунд.

Оперативна пам'ять є впорядкованим масивом однобайтових комірок, кожна з яких має свою унікальну адресу (номер). Типові

операції – читання і запис байта в комірку з потрібним номером. Обмін із зовнішньою пам'яттю зазвичай здійснюється блоками фіксованого розміру. Сукупність адрес у фізичній пам'яті називається фізичним адресним простором.

На жаль, багато термінів, що стосуються системи управління пам'яттю, як і в інформатиці взагалі, переобтяжені. Тому надалі термін "фізична пам'ять" відноситиметься саме до оперативної пам'яті, а використання зовнішньої пам'яті (дискової, файлів вивантаження) обмовлятиметься окремо.

Логічна пам'ять – абстракція, що відображає погляд користувача на те, як організовані його програми і зберігаються дані. З погляду користувача його виконувана програма (процес) є сукупністю блоків змінного розміру, що містять однорідну інформацію (дані, код, стек і т.д.). Зазвичай такі модулі називають *сегментами* (див. рис. 7.1).

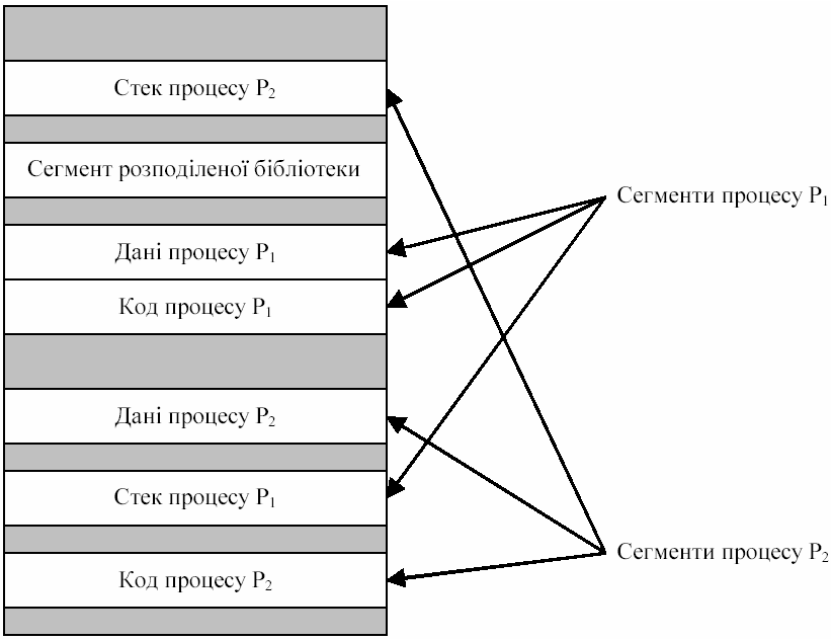


Рис. 7.1. Розташування сегментів процесів в пам'яті комп'ютера

Адреса при цьому перестає бути лінійною і складається з кількох компонентів, наприклад, номера сегмента і зміщення усередині сегмента. Крім того, із сегментами прийнято зв'язувати атрибути: права доступу або типи операцій, які дозволяється проводити з даними, що зберігаються в сегменті.

Логічні адреси усередині сегментів можуть бути сформовані на етапі компіляції. При цьому символічні імена зв'язуються з переміщуваними адресами (такими, як n байт від початку модуля). Іншим прикладом логічної адреси може бути адреса, одержана програмою в результаті операції виділення області пам'яті (allocation). Іноді говорять, що логічна адреса – це адреса, яку генерує процесор. Сукупність усіх логічних адрес називається *логічним (віртуальним) адресним простором*.

Скріплення адрес

Будучи віртуальною (абстрактною) машиною, ОС повинна привести у відповідність погляд користувача на організацію його програми з реальним зберіганням інформації у фізичній пам'яті. Ця проблема традиційно називається проблемою скріплення логічної і фізичної адрес (див. рис. 7.2). Вживаються також терміни „прив'язка адреси", „трансляція адреси", „дозволи адреси" і т.д. У ОС Windows це робиться на етапі виконання, тобто в момент звернення до логічної адреси менеджер пам'яті знаходить його візаві у фізичній пам'яті.

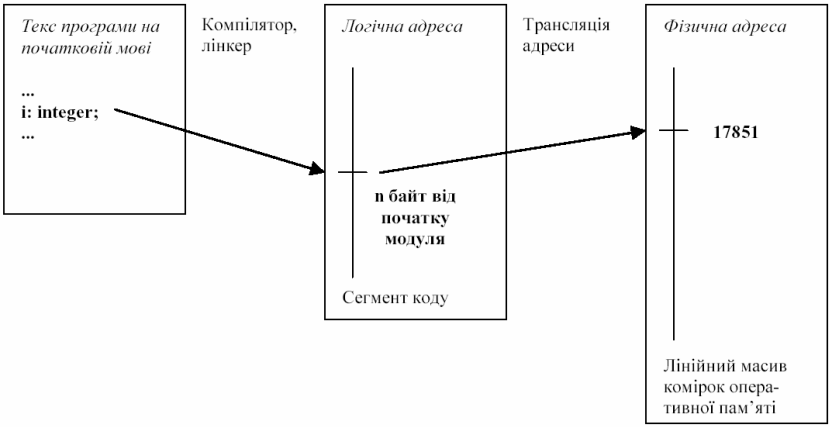


Рис. 7.2. Формування логічної адреси і скріплення логічної з фізичною

У сучасних обчислювальних системах типова ситуація, коли об'єм логічної пам'яті істотно перевищує об'єм оперативної. В цьому випадку логічна адреса може бути пов'язана з адресою в зовнішній пам'яті.

Розглянемо тепер алгоритми і структури даних, використовувані для опису логічної і фізичної пам'яті ОС Windows, а також вживану схему скріплення адрес. У якомусь сенсі віртуальна пам'ять є інтерфейсом системи управління пам'яттю, а її відображення у фізичну пам'ять і управління фізичною пам'яттю належать до особливостей реалізації.

Загальний опис віртуальної сегментно-сторінкової пам'яті ОС Windows

Розмір процесу користувача обмежений об'ємом логічного адресного простору. Характерний розмір логічної пам'яті визначається розрядністю архітектури і складає для сучасних систем 2^{32} (у недалекому майбутньому 2^{64}) байт. Ця величина зазвичай істотно перевищує об'єм оперативної пам'яті, тому частина процесу користувача прозорим чином може бути розміщена в зовнішній пам'яті. Тому в користувача створюється ілюзія того, що він має справу з віртуальною пам'яттю, відмінною від реальної, розмір якої потенційно більший, ніж розмір оперативної пам'яті. Надалі разом із терміном "логічна пам'ять" вживатиметься термін "віртуальна пам'ять".

Для визначення схеми віртуальної пам'яті, реалізованої в ОС Windows, найкраще підходить термін "сегментно-сторінкова віртуальна пам'ять". Докладний опис сегментно-сторінкової моделі можна знайти у відповідній літературі. Для неї характерне зображення адресного простору процесу у вигляді набору сегментів змінного розміру, що містять однорідну інформацію (дані, текст програми, стек, сегмент розподіленої пам'яті та ін.). Для зручності відображення на фізичну пам'ять кожен сегмент ділиться на сторінки – блоки фіксованого розміру, при цьому фізична пам'ять ділиться на блоки того ж розміру – сторінкові кадри (фрейми). Функція скріплення логічної адреси з фізичною покладається на таблицю сторінок, яка кожній логічній сторінці сегмента ставить у відповідність сторінковий кадр. У тих випадках, коли для по-

трібної сторінки не знаходиться місця в оперативній пам'яті (page fault), вона підкачується з диска. Відмітимо, що в канонічному вигляді даної схеми кожен сегмент процесу знаходиться в окремому логічному адресному просторі і використовує власну таблицю сторінок. Остання обставина, через складну організацію і великий об'єм таблиці сторінок, має наслідком той факт, що реальні системи рідко дотримуються канонічної форми.

Сегментно-сторінкова модель пам'яті, реалізована в ОС Windows, також має свою специфіку. Наприклад, апаратна підтримка сегментації, пропонована архітектурою Intel, використовується в мінімальному ступені, а такі фрагменти адресного простору процесу, як код, дані тощо, описуються за допомогою спеціальних структур даних і називаються регіонами (regions).

Одне із завдань, яке виконується при цьому, – уникнути появи в системі великої кількості таблиць сторінок за рахунок організації регіонів, що неперекриваються, в одному віртуальному просторі, для опису якого вистачає однієї таблиці сторінок. Таким чином, одна таблиця сторінок відводитиметься для всіх сегментів пам'яті процесу. Як саме це робиться, можна побачити на рис. 7.3. Задіяно всього чотири апаратні сегменти з номерами селекторів 08, 10, 1b і 23. Перший використовується для адресації коду ОС і має атрибути RE, другий з атрибутами RW – для даних і стека ОС, третій з атрибутами RE – для коду процесу користувача, а четвертий з атрибутами RW – для даних і стека процесу користувача. Перші два сегменти недоступні для непривілейованого режиму роботи процесора.

При цьому все організовано так, щоб використовувані віртуальні адреси всередині сегментів не перекривалися. В результаті виходить плоский 32-розрядний простір, що відображається на фізичну пам'ять за допомогою однієї дворівневої таблиці сторінок.

Цікаво, що наявність в апаратного сегмента атрибуту не є перешкодою для нецільового використання інформації, що зберігається в сегменті. Наприклад, код процесу, що знаходиться в сегменті 1b, може бути доступний через 23-й сегмент з атрибутами RW.

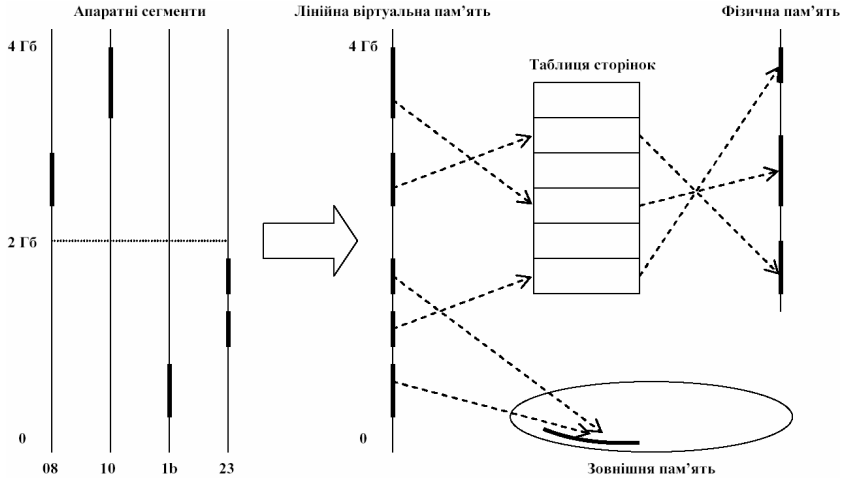


Рис. 7.3. Утворення регіонів (програмних сегментів), що неперекриваються, в лінійному віртуальному адресному просторі процесу

Власне захист регіонів організований на рівні їх описувачів, які зберігаються в таблиці описувачів VAD (virtual address descriptors) в адресному просторі процесу. Таким чином, апаратна підтримка сегментації забезпечує лише мінімальний захист – неможливість доступу до даних ОС із непривілейованого режиму. Можна сказати, що в ОС Windows здійснюється програмна підтримка сегментації (в даному випадку регіонів). Між іншим, багато інших ОС (наприклад, Linux) поводяться аналогічно. Програмна підтримка сегментів більш універсальна і сприяє більшій переносимості коду. Надалі для позначення безперервного фрагмента віртуального адресного простору, що містить однорідну інформацію, використовуватиметься термін "регіон".

Таблиця сторінок ставить у відповідність віртуальній сторінці номер сторінкового кадру в оперативній пам'яті. Для опису сукупності зайнятих і вільних сторінкових кадрів ОС Windows використовує базу даних PFN (page frame number). Через невідповідність розмірів віртуальної і оперативної пам'яті достатньо типовою є ситуація відсутності потрібної сторінки в оперативній пам'яті (page fault). На щастя, копії всіх задіяних віртуальних сторінок зберігаються на диску (так звані тіньові сторінки). У разі

звернення до відсутньої сторінки ОС повинна розшукати відповідну тіньову сторінку і організувати її підкачку з диска.

Облік сукупності тіньових сторінок пов'язаний із труднощами, зумовленими розрідженістю використовуваних віртуальних адрес. Так, незмінні сторінки коду програми беруться безпосередньо з виконуваних файлів (техніка – відображення файла, що містить код програми, в пам'ять). Сторінки, схильні до змін, періодично записуються в спеціальні файли вивантаження.

Таким чином, діяльність системи управління пам'яттю зводиться до створення регіонів (програмних сегментів) у віртуальному адресному просторі, виділення для них місця у фізичній пам'яті (частково в оперативній пам'яті і частково на диску) і прозорого перенаправлення звернень від віртуальних адрес до їх аналогів у фізичній пам'яті. Регіони створюються операційною системою. Іноді це відбувається за ініціативою програми користувача (наприклад, у результаті виклику функцій VirtualAlloc, CreateFileMapping, CreateHeap та ін.). Істотна частина діяльності менеджера пам'яті пов'язана з оптимізацією. Зокрема, багато зусиль витрачається на скорочення кількості звернень до зовнішньої пам'яті.

Перейдемо тепер до детальнішого розгляду описаної схеми. Спочатку вивчимо віртуальний адресний простір процесу, потім подивимося, як ключова інформація розміщується у фізичній пам'яті. Проблема скріплення адрес розв'язується, головним чином, за рахунок апаратних засобів архітектури, тому ці питання зачіпатимуться в міру необхідності.

Інструментальні засоби спостереження за роботою менеджера пам'яті

З інструментальних засобів Windows, описаних, для кращого практичного ознайомлення з діяльністю по управлінню пам'яттю в роботі, активно використовуватимуться вкладки "Процеси" і "Швидкодія" диспетчера завдань, а також різноманітні лічильники продуктивності, за поведінкою яких можна стежити з оснащення "Продуктивність" ("Системний монітор") адміністративної консолі панелі управління.

Найцікавішу інформацію містять лічильники, що відносяться до конкретного процесу: кількість байтів віртуальної пам'яті, файла підкачки, помилок сторінки, робочої множини і т.д., і загальносистемні лічильники: кількість байтів виділеної віртуальної пам'яті, елементів таблиці сторінок та ін.

Корисними виявляються і деякі загальнодоступні утиліти, наприклад утиліта спостереження за помилками сторінок pfmon.

Віртуальний адресний простір процесу

У 32-бітових системах процесор може згенерувати 32-бітну адресу. Це означає, що кожному процесу виділяється діапазон віртуальних адрес від 0x00000000 до 0xFFFFFFFF. Ці 4 Гб адрес система ділить приблизно навпіл, для коду і даних режиму користувача відводяться 2 Гб у нижній частині пам'яті. Якщо бути точнішим, то йдеться про віртуальні адреси, починаючи з 0x00010000 і закінчуючи 0x7FFEFFFF. Таким чином, система управління пам'яттю дозволяє програмі користувача за допомогою Win32 API записати потрібний байт у будь-яку віртуальну комірку з цього діапазону адрес. Адреси верхньої частини віртуальної пам'яті використовуються для коду і даних режиму ядра та інших системних потреб.

За замовчуванням адресний простір кожного процесу ізольований. Дані двох різних процесів, записані по одній і тій же віртуальній адресі, опиняються в різних сторінках фізичної пам'яті за допомогою коректної роботи системи трансляції адреси. В ряді випадків ізоляція може бути частково знята (файли, що відображаються в пам'ять; пам'ять, що розділяється). Зрозуміло, в подібних випадках потрібно окремо забезпечити контроль доступу до області пам'яті, для чого створюється окремий об'єкт (об'єкт-секція або об'єкт-розділ, section object), що включає атрибути захисту. Нижче буде також наведений приклад контролю процесом пам'яті іншого процесу – прийом, яким активно користуються відладчики.

Регіони у віртуальному адресному просторі

Спочатку весь віртуальний адресний простір процесу вільний. Він починає заповнюватися в міру виконання програми. Щоб скористатися якою-небудь частиною цього простору, в ньому по-

трібно створити регіон, зарезервувавши певний діапазон адрес. Така схема дозволяє підтримувати розріджені адресні простори.

Сукупність регіонів описується структурою VAD, яка організована у вигляді двійкового дерева і зберігається в PCB (структура EPROCESS, див. лабораторну роботу № 3) процесу. Для новостворюваного регіону можна запитати будь-який діапазон адрес з урахуванням уже існуючих регіонів. Перші регіони для коду, стека, стандартної купи процесу і ряд інших створює операційна система у момент завантаження процесу. Подальші регіони додаток створює самостійно (див. рис. 7.4).

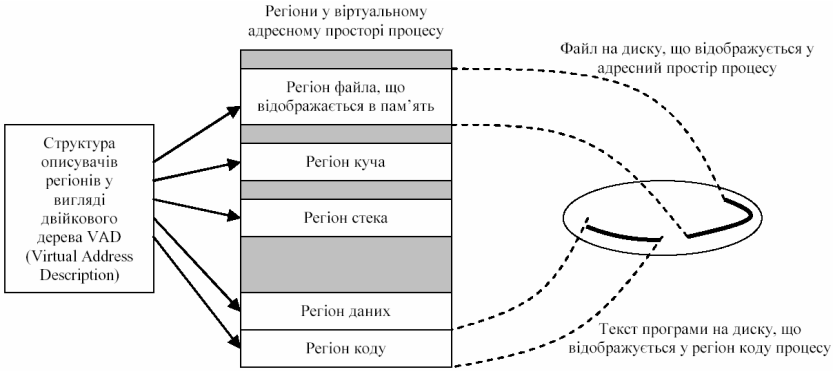


Рис. 7.4. Сукупність регіонів у частині користувача (нижні 2 Гб) віртуального адресного простору процесу

Створення (резервування) регіону і передача йому фізичної пам'яті

Для створення регіону явним чином зазвичай використовується функція VirtualAlloc (виклик ряду Win32 функцій, таких як CreateFileMapping або CreateHeap, також має наслідком створення регіону). В процесі створення регіону виділяють два етапи: резервування регіону і передачу йому фізичної пам'яті (commit). Обидва етапи виконуються в результаті виклику VirtualAlloc і можуть бути об'єднані. У результаті кожна віртуальна сторінка може опинитися в одному з трьох станів: вільна (free), зарезервована (reserve) і передана (committed)

Резервування регіону – швидка операція, а в подальшій передачі пам'яті не завжди виникає необхідність; таким чином, двохетапний процес виділення дозволяє підвищити ефективність системи управління пам'яттю.

Резервування регіону передбачає вирівнювання початку регіону з урахуванням гранулярності пам'яті (зазвичай це 64 Кб). Крім того, розмір регіону повинен бути кратний об'єму сторінки (4Кб для x86 процесора. Дізнатися розмір сторінки можна за допомогою функції GetSystemInfo). У разі успішного резервування відбувається корекція дерева VAD.

Для створюваного регіону можна вказати конкретний діапазон віртуальних адрес. Якщо цього не робити, то система проглядає віртуальний адресний простір процесу, намагаючись знайти безперервну незарезервовану область потрібного розміру.

Щоб використовувати зарезервований регіон, йому потрібно передати, тобто реально виділити, фізичну пам'ять (фізичну або зовнішню по розсуду ОС). Немає необхідності передавати фізичну пам'ять всьому регіону цілком. Більше того, це рекомендується робити поетапно, в міру необхідності. Так забезпечується економія фізичної пам'яті. Наприклад, складні додатки, що працюють із великими масивами даних, використовують таку стратегію: спочатку фізична пам'ять не передається, як тільки відбувається звернення до віртуальної адреси, під яку не виділена пам'ять, вона одразу ж виділяється. Як правило, подібні ситуації обробляються за допомогою структурної обробки виключень.

Дослідження роботи програми виділення пам'яті за допомогою функції VirtualAlloc

У програмі, текст якої наведений нижче, здійснюється резервування регіону розміром 16 сторінок віртуальної пам'яті і потім передача фізичної пам'яті його четвертої сторінки. У верхню частину переданої пам'яті записується рядок, який потім роздруковується. Легко переконатися, що вихід за межі переданої пам'яті за допомогою змінної Shift призводить до помилки виконання. В кінці регіон звільняється за допомогою функції VirtualFree.

Приклад 7.1

```
uses windows;
var pMem:pointer=NIL;
   String_,pMemCommited:^char;
   nPageSize:dword=4096;
   Shift:dword=4080;
   s:string='Hello, world';
   i:integer;
begin
pMem:=VirtualAlloc(NIL,nPageSize*16,MEM_RESERVE,
                                          PAGE_READWRITE);
pMemCommited:=pMem + 3*nPageSize;
VirtualAlloc(pMemCommited,nPageSize,MEM_COMMIT,
                                          PAGE_READWRITE);
String_:=pMemCommited + Shift;
wsprintf(String_,(@s)+1);
for i:=1 to length(s) do begin write(String_^);inc(String_);end;
VirtualFree(String_,0,MEM_RELEASE);
end.
```

Дослідження роботи програми, що демонструє виділення великих масивів пам'яті

Розглянемо наступний приклад DemoVM.pas. У програмі в декілька етапів по натисненню клавіші "Enter" виділяються і передаються регіонам великі масиви фізичної пам'яті. Необхідно здійснити спостереження за виділенням пам'яті процесу за допомогою лічильника "Байт віртуальної пам'яті, виділеної процесу".

Приклад 7.2

```
uses windows;
var pMem:pointer=NIL;
   nPageSize:dword=4096;
   SizeCommit:dword=0;
   nPages:dword=200;
begin
SizeCommit:=nPages*nPageSize; readln;
pMem:=VirtualAlloc(NIL,SizeCommit,MEM_RESERVE of
                          MEM_COMMIT, PAGE_READWRITE);
```

```
if pMem = NIL then Writeln('VirtualAlloc Error'); readln;
pMem:=VirtualAlloc(NIL,SizeCommit,MEM_RESERVE of
                              MEM_COMMIT, PAGE_READWRITE);
if pMem = NIL then Writeln('VirtualAlloc Error'); readln;
pMem:=VirtualAlloc(NIL,SizeCommit,MEM_RESERVE or
                              MEM_COMMIT, PAGE_READWRITE);
if pMem = NIL then Writeln('VirtualAlloc Error'); readln;
pMem:=VirtualAlloc(NIL,SizeCommit,MEM_RESERVE or
                              MEM_COMMIT, PAGE_READWRITE);
if pMem = NIL then Writeln('VirtualAlloc Error'); readln; end.
```

З тексту програми видно, що кожного разу по натисненню клавіші "Enter" процесу передається 200 сторінок (819200 байт) віртуальної пам'яті. Це легко перевірити по відповідному приросту лічильника "Байт віртуальної пам'яті" (див. рис. 7.5).

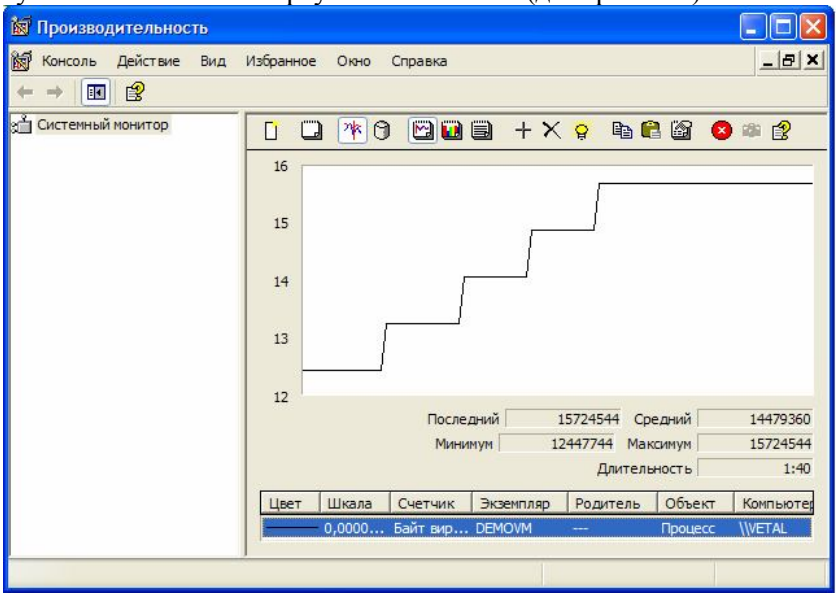


Рис. 7.5. Поведінка лічильника "Байт віртуальної пам'яті", виділеної процесу DemoVM

Як самостійну вправу можна рекомендувати дослідження роботи даної програми з іншими параметрами.

Регіон купа

Як уже мовилося, в ряді випадків система сама резервує регіони в адресному просторі процесу. Прикладами таких регіонів можуть служити регіон купа і регіон стека потоку.

Купа (heap) – зарезервований регіон розміром в одну і більш сторінок, який рекомендується використовувати для зберігання множини невеликих порцій даних. На відміну від функції VirtualAlloc, правила гранулярності виділення пам'яті при цьому дотримуватися не обов'язково. Передачею пам'яті купам, а також обліком вільної і зайнятої пам'яті в купі займається спеціальний диспетчер куп (heap manager). Ця діяльність не документована.

Стандартна купа процесу розміром 1 Мб (цю величину можна змінити) резервується у момент створення процесу. Зазвичай купа динамічно змінює свій розмір (прапор growable). Стандартну купу процесу використовують не тільки додатки, але і деякі Win32-функції. Для використання стандартної купи необхідно одержати її описувач за допомогою функції GetProcessHeap.

При бажанні процес може створити додаткові купи за допомогою функції HeapCreate (зворотна операція HeapDestroy). Прикладна програма виділяє пам'ять у купі за допомогою функції HeapAlloc, а звільняє за допомогою HeapFree. Оскільки купами можуть користуватися всі потоки процесу, за замовчуванням організовується синхронізація (прапор SERIALIZE), яку, хоча це і не рекомендується, для підвищення швидкодії можна відмінити. Питання синхронізації доступу потоків до купи можна також вирішити, створюючи кожному потоку власну купу.

Дослідження роботи програми виділення пам'яті в стандартній купі

Приклад 7.3

```
uses windows;
var hHeap:HANDLE;
   String_,pHeap:^char;
   Size_:dword=1024;
   Shift:dword=1016;
   s:string='Hello, world';
   i:integer;
```

```
begin
hHeap:=GetProcessHeap;
pHeap:=HeapAlloc(hHeap,0,Size_);
if(pHeap=NIL)then writeln('HeapAlloc error') else
String_:=pHeap+Shift;
wsprintf(String_,(@s)+1);
write('Heap contents string: ');
for i:=1 to length(s) do begin write(String_^);inc(String_);end;
HeapFree(hHeap,0,pHeap);  end.
```

У вищенаведеній програмі відбувається виділення масиву пам'яті в стандартній купі процесу. Далі туди записується текстовий рядок, який потім виводиться на екран. Якщо виникає ситуація виходу за межі виділеної пам'яті, яку легко змоделювати, збільшуючи значення параметра Shift, – виникає помилка виконання.

Як самостійну вправу рекомендується спостереження за нарощуванням об'єму переданої в купі пам'яті за допомогою лічильників продуктивності. Об'єм віртуальної пам'яті процесу повинен почати рости у випадку виходу за межі початкового розміру купи (1 Мб) і при утворенні додаткових куп.

Регіон стека потоку. Сторожові сторінки

Для підтримки функціонування стека потоку також резервується відповідний регіон. Стек – динамічна структура. Прийнято, щоб стек збільшував свій розмір у бік зменшення адрес. Скільки сторінок пам'яті буде потрібно стеку потоку, наперед не відомо. Тому в ОС Windows організована поетапна, в міру необхідності, передача фізичної пам'яті стека за допомогою механізму так званих сторожових сторінок (guard page). Звернення до сторожової сторінки має наслідком повідомлення системи про це (виняткова ситуація 0x80000001), після чого прапор PAGE_GUARD скидається і зі сторінкою можна працювати як зі звичайною сторінкою пам'яті. Сторожова сторінка служить пасткою для перехоплення посилань за її межі.

При створенні потоку для його стека резервується регіон розміром 1 Мб (за замовчуванням), і йому передається 2 сторінки пам'яті (ці параметри можуть бути змінені). Нижня сторінка є

сторожовою. Як тільки верхня сторінка виявилася заповненою і відбулося звернення до нижньої сторінки, це помічається системою і регіону передається ще одна сторінка, тепер уже вона стає сторожовою. Внаслідок такої тактики найнижча передана регіону стека сторінка завжди залишається сторожовою і її завдання – просигналізувати системі про те, що об'єм переданої стеку пам'яті потрібно збільшити.

Дослідження роботи програми, що моделює звернення до сторожових сторінок

У наведеній у прикладі 7.4 програмі відбувається передача пам'яті регіону і встановлення прапора PAGE_GUARD для його сторінок. Окрім того, використовується обробка виключень новою функцією обробника виключень new_filter, яка за допомогою API-функції SetUnhandledExceptionFilter замінює функцію-фільтр системного обробника виключень. При виникненні виключної ситуації програмним шляхом за допомогою функції RaiseException (у основній програмі), відбувається спроба запису текстового рядка на сторожову сторінку (у функції обробника виключень new_filter) і виникає виняткова ситуація exception 0x80000001 (або -2147483647). При повторній спробі запису на цю сторінку (при рекурсивному виклику функції new_filter) подібна ситуація вже не виникає.

Приклад 7.4

```
uses windows;
var hThread:HANDLE;
   pMem:pointer=NIL;
   s_m,String_,pMemCommited:^char;
   nPageSize:dword=4096;
   Shift:dword=4000;
   s:string='Hello, world';
   i,j:integer=0;
   old_filter:TFNTopLevelExceptionFilter;
   Exc_Cod:longint;
   mes:array[1..2]of string;
function
new_filter(pExceptionlnfo:TEXCEPTIONPOINTERS):longint;
```

```
begin  s_m:=String_;
wsprintf(String_,(@s)+1);
Exc_Cod:=pExceptionlnfo.ExceptionRecord^.ExceptionCode;
inc(j);Write(mes[j],Exc_Cod,', string: ');
for i:=1 to length(s) do begin write(String_^);inc(String_);end;
String_:=s_m;writeln;
new_filter:=-1;//EXCEPTION_CONTINUE_EXECUTION end;
begin
mes[1]:='Before exception number: ';
mes[2]:='After exception number: ';
old_filter:=SetUnhandledExceptionFilter(@new_filter);
pMem:=VirtualAlloc(NIL,nPageSize*16,MEM_RESERVE or
MEM_COMMIT,PAGE_READWRITE or PAGE_GUARD);
pMemCommited:=pMem+3*nPageSize;
String_:=pMemCommited+Shift;
RaiseException(0,0,0,nil);
VirtualFree(String_, 0, MEM_RELEASE);
SetUnhandledExceptionFilter(old_filter); end.
```

Як самостійну вправу рекомендується написати програму, в якій сторожова сторінка весь час знаходиться на межі фрагмента переданої пам'яті. У випадку звернення до сторожової сторінки об'єм переданої процесу пам'яті потрібно збільшити, а сторожову сторінку перемістити.

Регіон файла, що відображається в пам'ять

Техніка файлів, що проектуються в пам'ять (див. рис. 7.4), активно використовується новітніми ОС. Вона дозволяє користувачу розв'язувати такі задачі, як робота з даними файла за допомогою операцій копіювання і переміщення байтів у пам'яті або організація сумісного доступу до областей пам'яті. Цей механізм також активно використовується самою операційною системою, наприклад для завантаження в пам'ять виконуваних модулів, динамічних бібліотек і відображення файла в буфер кешу для здійснення операцій стандартного введення-виведення.

Відображення файла в пам'ять означає резервування регіону потрібного розміру і передача йому відповідного об'єму фізичної пам'яті (передавати пам'ять тут теж можна поетапно). Проте, на

відміну від звичайних регіонів, вивантаження фрагментів оперативної пам'яті в зовнішню пам'ять при цьому здійснюватиметься не в системну область зовнішньої пам'яті (pagefile.sys), а безпосередньо у файл, що відображається. Відображення може бути виконане на конкретний діапазон віртуальних адрес, якщо це не суперечить місцеположенню вже існуючих регіонів віртуальної пам'яті.

Для відображення файла в пам'ять використовується функція CreateFileMapping, а для отримання покажчика на відображену область – функція MapViewOfFile. Успішне виконання обох операцій дозволяє прикладній програмі працювати з цією областю як із будь-яким іншим фрагментом виділеної пам'яті, зокрема змінювати її вміст. У зв'язку з цим виникає проблема відповідності (когерентності) вмісту регіону і файла на диску. Операційна система прагне забезпечити когерентність, проте у розпорядженні користувача є можливість у будь-який момент скинути вміст пам'яті на диск за допомогою функції FlushViewOfFile.

Інший цікавий момент пов'язаний із тим, що система розглядає проектований у пам'ять файл як об'єкт багатоцільового призначення. Тому відображення файла в пам'ять супроводжується створенням супутнього об'єкта ядра (в даному випадку це об'єкт-секція) з ім'ям у просторі об'єктів, по якому він може бути доступний іншим процесам, лічильникам посилань з атрибутами захисту. Як завжди, посилання на об'єкт зберігається в таблиці описувачів кожного процесу, що має доступ до об'єкта.

Дослідження роботи програми, що демонструє відображення файла в пам'ять

Наведені у прикладах 7.5 та 7.6 програми демонструють етапи створення файла, проектування його в пам'ять, зміни його вмісту і відображення на диск із використанням функції FlushViewOfFile. Перша програма створює файл, потім відображає його в пам'ять та змінює його вміст. Після цього запускається друга програма, яка зчитує вміст відображеного в пам'ять файла. Для того, щоб друга програма зчитувала змінений файл, його вміст перед запуском другої програми записується на диск за допомогою виклику функції FlushViewOfFile.

Приклад 7.5

```
uses windows;
var a:array[0..9]of integer;
   file_name:string=‘Demo.bin’;
   mapping_name:string=‘MappingName’;
   hFile,hMapping:HANDLE;
   p,ptr_:^integer;
   fp:file of integer;
   i:integer;
   IpszAppName:string=‘CON_PROC.EXE’;
   si:TSTARTUPINFO;
   piApp:TPROCESSINFORMATION;
begin    for i:=0 to 9 do a[i]:=i;
        assign(fp,file_name);
        rewrite(fp);
        write(‘Initial array: ‘);
        for i:=0 to 9 do begin
        write(fp,a[i]);
        write(a[i],’,’);  end;
        close(fp);
hFile:=CreateFile((@file_name)+1,GENERIC_READ or
GENERIC_WRITE,FILE_SHARE_READ or
FILE_SHARE_WRITE,NIL,OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL,0);
hMapping:=CreateFileMapping(hFile,NIL,
PAGE_READWRITE,0,0,(@mapping_name)+1);
ptr_:=MapViewOfFile(hMapping,FILE_MAP_WRITE,0,0,0);
for i:=0 to 9 do begin new(p);p:=ptr_+i*4;p^:=p^+10;end;
FlushViewOfFile(ptr_,0);
ZeroMemory(@si,sizeof(si));
si.cb:=sizeof(si);
CreateProcess((@IpszAppName)+1,NIL,NIL,NIL,FALSE,
          CREATE_NEW_CONSOLE,NIL,NIL,si,piApp);
WaitForSingleObject(piApp.hProcess,INFINITE);
CloseHandle(piApp.hThread);
CloseHandle(piApp.hProcess);
```

```
UnmapViewOfFile(ptr_);
CloseHandle(hMapping);
CloseHandle(hFile); readln; end.
```

Приклад 7.6

```
uses Windows;
var a:array[0..9]of integer;
   file_name:string='Demo.bin';
   fp:file of integer;
   i:integer;
begin   assign(fp,file_name);
        reset(fp);
        write('Final array: ');
        for i:=0 to 9 do begin
        read(fp,a[i]);
        write(a[i],',');  end;
        close(fp);
        readln;  end.
```

На основі попередньої програми рекомендується написати програму відображення файла в пам'ять із проміжним вивантаженням файла на диск за допомогою функції FlushViewOfFile. Розгляньте різні варіанти існування файла і його розмірів до і після відображення.

Розв'язання задачі сумісного доступу до пам'яті буде наведено в наступних лабораторних роботах.

Тут ми тимчасово припинимо вивчення віртуальної пам'яті процесу. Основним підсумком викладеного можна вважати знайомство з можливостями ОС створювати в ній різноманітні регіони. Як самостійну вправу можна рекомендувати аналіз стану віртуального адресного простору за допомогою функцій VirtualQuery і VirtualQueryEx.

Висновок

Таким чином, система управління пам'яттю є однією з найбільш важливих у складі ОС. Традиційна схема припускає скріплення віртуальної і фізичної адреси на стадії виконання програми. Для управління віртуальним адресним простором у ньому прийнято організовувати сегменти (регіони), для опису яких ви-

користовуються структури даних VAD (virtual address descriptors). Для створення регіону і передачі йому фізичної пам'яті можна використовувати функцію VirtualAlloc. Описана техніка використання таких регіонів, як купа процесу, стек потоку і регіон файла, що відображається в пам'ять.

Практична частина

1. Складіть програми, наведені у прикладах 7.1 – 7.6 теоретичної частини даної лабораторної роботи. Запустіть програми та переконайтеся, що вони працюють правильно (згідно з описом алгоритму їх роботи).

2. На основі наведених програм дослідіть різноманітні особливості функціонування системи керування пам'яттю операційної системи Windows.

3. Оформіть звіт із виконаної лабораторної роботи, який повинен містити текст програм, демонстрацію результатів роботи, та дайте відповіді на контрольні запитання.

Контрольні запитання та завдання

1. В чому полягає різниця між фізичною пам'яттю та логічною?
2. Опишіть процес скріплення логічної адреси з фізичною.
3. Що таке менеджер пам'яті та яка його функція?
4. Опишіть основні особливості віртуальної сегментно-сторінкової організації пам'яті операційної системи Windows.
5. Розкрийте поняття регіону.
6. Опишіть особливості організації регіонів у віртуальному адресному просторі процесу.
7. Обґрунтуйте необхідність організації механізму "сторожових" сторінок у регіоні стека потоку.
8. Яке призначення регіону „купа" (heap) та які API-функції керування ним ви знаєте?
9. В яких випадках відбувається запис у зовнішню пам'ять регіону файла, що відображається в основну пам'ять?

Лабораторна робота № 8
Дослідження особливостей функціонування менеджера пам'яті операційної системи Windows

Теоретична частина

Після знайомства зі структурою віртуального адресного простору в попередній лабораторній роботі перейдемо до розгляду питань скріплення віртуальних адрес із фізичними адресами в рамках сегментно-сторінкової моделі пам'яті Windows.

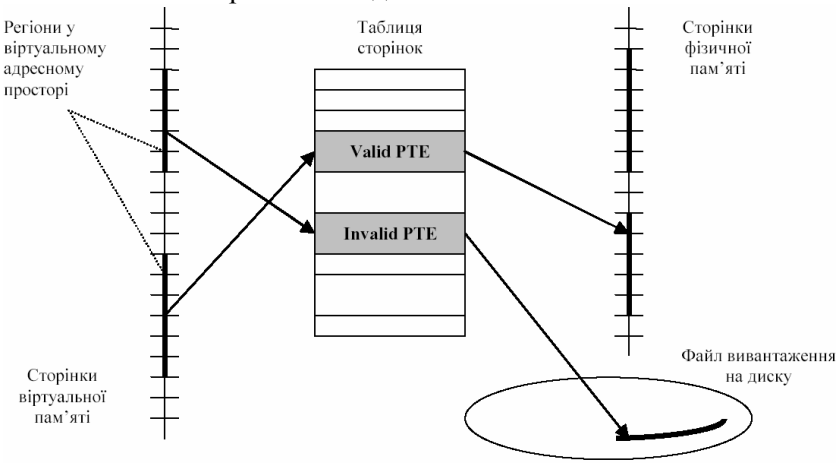


Рис. 8.1. Трансляція адреси в сторінковій віртуальній схемі

На рис. 8.1 подано відображення різних типів віртуальних адрес у фізичні. Для трансляції віртуальної адреси, згенерованої процесором, потрібно визначити номер віртуальної сторінки, що містить дану адресу, а також приналежність сторінки до зарезервованого регіону. Деяким віртуальним сторінкам таблиця сторінок ставить у відповідність фізичні сторінки (сторінкові кадри або фрейми) в оперативній пам'яті. Цікаво, що код режиму ядра, наприклад драйверів, може виконати зворотну трансляцію – отримання віртуальної адреси за фізичною. Тим сторінкам, яким місця у фізичній пам'яті не знайшлося, відповідають тіньові сторінки в системному файлі вивантаження. Таблиця сторінок тут

зображена схематично, її точніша версія буде продемонстрована нижче.

Кожен рядок у таблиці сторінок називається PTE (page table entry). У 32-розрядній архітектурі IA-32 PTE займає 4 байти (див. рис. 8.2).

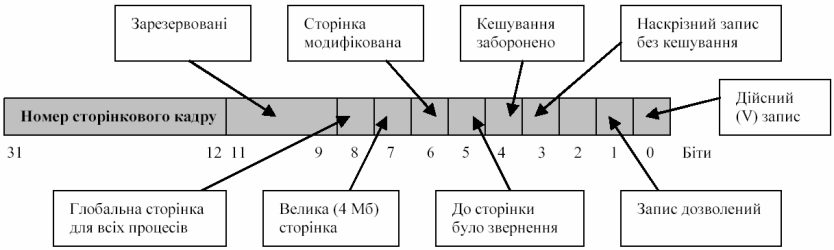


Рис. 8.2. Структура рядка таблиці сторінок (PTE)

Призначення окремих атрибутів зрозуміле з рисунка. Наприклад, інформація про звернення до сторінки або її модифікації (біти 5 і 6) дозволяє збирати статистику звернень до пам'яті, яку використовують алгоритми виштовхування сторінок. Із погляду процесу трансляції найбільш важливу роль відіграє біт присутності V (Valid). Згідно з прийнятою термінологією PTE зі встановленим бітом V називаються "дійсними", а відповідна сторінка знаходиться в оперативній пам'яті.

При скинутому V біті виникає *сторінкова помилка (page fault)*. Найбільш поширений варіант page fault – знаходження сторінки у файлі вивантаження (відсутня сторінка). В цьому випадку перші 20 бітів PTE вказуватимуть на зсув у сторінковому файлі. Обробка таких ситуацій полягає в припиненні процесу, що згенерував page fault, підкачці сторінки з диска у вільний кадр фізичної пам'яті, модифікації PTE і відновленні невдалої операції. Таким чином, недійсний PTE перетворюється на дійсний. Ця ситуація виникає часто, більше того, нерідкі випадки звернення до однієї і тієї ж відсутньої сторінки двох потоків одного процесу (конфлікт помилок сторінок), які успішно обробляються системою. Крім потрібної сторінки, диспетчер про всяк випадок завантажує в пам'ять декілька, зазвичай від 1 до 8, сусідніх сторінок,

щоб мінімізувати кількість звернень до диска (стратегія підкачки на вимогу з кластеризацією).

Насправді трансляція відбувається складніше. Істотна частина записів PTE кешується в асоціативній пам'яті (TLB регістрах) процесора. При цьому таблицю сторінок через її великий об'єм конструюють із двох рівнів: каталог таблиць сторінок (page directory) і таблиці сторінок (page table), див. рис. 8.3.

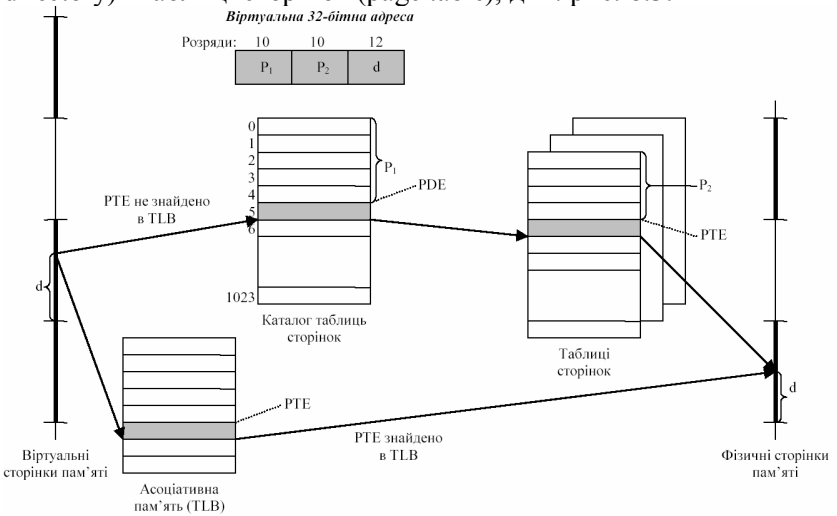


Рис. 8.3. Трансляція адреси з використанням асоціативної пам'яті і дворівневої таблиці сторінок

Розмір таблиці сторінок підібраний так, що вона цілком заповнює одну сторінку оперативної пам'яті – 4 Кб. Для швидкого знаходження таблиці сторінок один із регістрів процесора (CR3 у Intel) вказує на каталог таблиць сторінок (page directory) даного процесу, який зберігається за адресою 0xC0300000. Значення цього регістра входить у контекст процесу. В зв'язку з цим, а також тому, що при зміні виконуваного потоку буфер асоціативної пам'яті потребує оновлення, дещо збільшується час перемикання контекстів. Ці особливості архітектури добре відомі і тут не описуватимуться детально. Надалі для спрощення в схемі трансляції адреси таблиця сторінок зображатиметься одновимірною.

Пам’ять, що розділяється

Як уже було зазначено в попередній лабораторній роботі, при відображенні файла в пам’ять утворюється регіон у віртуальній пам’яті, а також супутній йому об’єкт-розділ або об’єкт-секція (section object). Як і інші об’єкти, об’єкти-розділи управляються диспетчером об’єктів. У разі виникнення помилок сторінок підкачка здійснюється зі сторінок проектованого файла, а не із загальносистемного файла вивантаження.

Для таких регіонів формується масив прототипних PTE, що описують знаходження всіх сторінок цього фрагмента пам’яті. PTE таблиці сторінок процесу, що виконав відображення, посилаються на прототипні PTE, як це показано на рис. 8.4, і теж вважаються недійсними. Якщо при цьому інший процес виконав відображення цього ж файла, то PTE його таблиці теж посилатимуться на ці ж самі прототипні PTE (див. рис. 8.4). Таким чином, фізичні сторінки пам’яті будуть узагальненими.

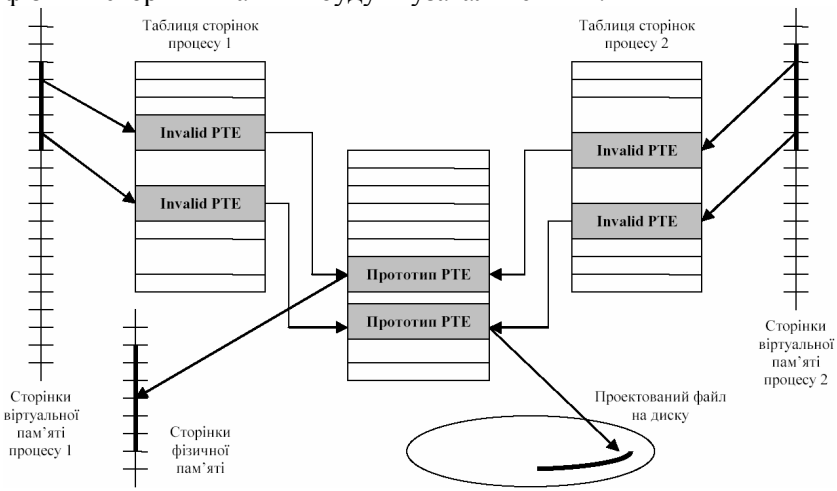


Рис. 8.4. Реалізація регіону проектованого в пам’ять файла, що розділяється між двома процесами

Зміни, зроблені в даному фрагменті пам’яті одним процесом, будуть відразу ж "видно" іншому процесу. Таким чином, наявність таблиці прототипних PTE забезпечує когерентність пам’яті, що розділяється.

Більше того, якщо третій процес відкрив той самий файл для звичайного введення-виведення, то вже існуюче відображення розглядатиметься як буфер кешу цього файла, доступ до якого здійснюватиметься через ту ж саму таблицю прототипних PTE. Отже, всі три процеси працюватимуть з однією і тією ж версією файла.

Крім обміну інформацією між різними процесами, сторінки, що розділяються, застосовуються також для передачі даних між користувацькою і ядерною частинами адресного простору процесу. Прикладами можуть служити передача системної інформації або передача інформації від драйвера введення-виведення.

Дослідження роботи програми, що демонструє передачу інформації від одного процесу до іншого через пам'ять, що розділяється

Розглянемо текст двох програм first.pas і second.pas

Приклад 8.1. (First.pas)

```
uses windows;
var hMapFile,hFile:HANDLE;
   lpMapAddress:PVOID;
   String_:^char;
   s:string='Hello, world';
begin  hFile:=CreateFile('MyFile.txt',GENERIC_READ or
GENERIC_WRITE, FILE_SHARE_READ or
FILE_SHARE_WRITE, NIL,OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL,0);
if(hFile=INVALID_HANDLE_VALUE)then
                                        writeln('Could not open file');
hMapFile:=CreateFileMapping(hFile,NIL,PAGE_READWRITE,0,0,
                                                  'MyFileObject');
if(hMapFile=0)then writeln('Could not create file-mapping object');
lpMapAddress:=MapViewOfFile(hMapFile,
                                    FILE_MAP_ALL_ACCESS,0,0,0);
if(lpMapAddress=NIL)then writeln('Could not map view of file');
String_:=lpMapAddress; wsprintf(String_,(@s)+1); readln; end.
```

Приклад 8.2. (Second.pas)

```
uses windows;
var hMapFile,hFile:HANDLE;
   lpMapAddress:PVOID;
   String_:^char;
begin
hFile:=CreateFile('MyFile.txt',GENERIC_READ or
GENERIC_WRITE,FILE_SHARE_READ or
FILE_SHARE_WRITE,NIL,OPEN_EXISTING,
                                    FILE_ATTRIBUTE_NORMAL,0);
if(hFile=INVALID_HANDLE_VALUE)then
                                          writeln('Could not open file');
hMapFile:=OpenFileMapping(FILE_MAP_ALL_ACCESS,False,
                                                 'MyFileObject');
if(hMapFile=0)then writeln('Could not open Filemapping');
lpMapAddress:=MapViewOfFile(hMapFile,
                                    FILE_MAP_ALL_ACCESS,0,0,0);
if(lpMapAddress=NIL)then writeln('Could not map view of file');
String_:=lpMapAddress;
while(String_^<>chr(0))do
begin write(String_^);inc(String_);end;
readln;  end.
```

Програма first створює в своєму адресному просторі буфер пам'яті, що розділяється, а програма second відображає той самий буфер у свій адресний простір. Потім програма first записує в цей буфер текстовий рядок, а програма second виводить її вміст на екран. Обидві програми повинні бути запущені з одного каталогу з вже існуючим файлом MyFile.txt. Для наочності рекомендується, щоб довжина файла була спочатку більше від довжини рядка "Hello, world".

Як самостійну вправу рекомендується модифікувати попередню програму для передачі інформації через фрагмент пам'яті, що розділяється, спроектованої не в звичайний файл, а в системну область вивантаження. Для цього параметром описувача файла функції CreateFileMapping потрібно вказати INVALID_HANDLE_VALUE.

Фізична пам’ять

Фізична (в даному випадку оперативна) пам’ять і зовнішня пам’ять також описуються відповідними структурами даних.

ОС Windows підтримує до 4 Гб (деякі версії і більш) фізичної пам’яті. Пам’ять більше 32 Мб вважається "великою". Об’єм пам’яті можна подивитися на вкладці "Швидкодія" диспетчера завдань. Інформація про стан сторінок фізичної пам’яті і їх приналежності процесам знаходиться в базі даних PFN (page frame number), а використання зовнішньої пам’яті здійснюється через сторінкові файли або файли вивантаження.

Сторінкові файли, на відміну від файлів, що проектуються в пам’ять, зберігають тільки модифіковані сторінки, які з яких-небудь причин вивантажені на диск. Сторінки, що містять тексти програм, відображаються в пам’ять безпосередньо з виконуваних модулів і не зберігаються в загальносистемних файлах вивантаження.

Структура системних сторінкових файлів не документована. Відомо, що в системі може бути до 16 сторінкових файлів. Інформація про сторінкові файли знаходиться в розділі HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagingFiles реєстру, проте управління сторінковими файлами рекомендується здійснювати через аплет "система" адміністративної консолі управління. У кожного файла підкачки є початковий і максимальний розміри. З метою зменшення ймовірної фрагментації їх створюють максимального розміру.

Корисну інформацію про використання сторінкових файлів можна одержати, спостерігаючи за лічильниками на вкладці "Продуктивність", а також за допомогою диспетчера завдань. Наприклад, лічильник "Page File Bytes" показує загальну кількість переданих сторінок.

Робочі набори процесів

У результаті скріплення адрес частина віртуальних сторінок процесу безпосередньо відображається в сторінки фізичної пам’яті. Цю множину сторінок іноді називають *резидентною множиною* процесу. У теорії операційних систем відоме також

поняття "робочої множини" процесу – сукупності сторінок, що активно використовуються разом, що дозволяє процесу протягом деякого періоду часу продуктивно працювати, уникаючи великої кількості page fault.

Згідно з документацією по ОС Windows, робочим набором процесу називається сукупність фізичних сторінок, виділених процесу. Розмір робочого набору повинен знаходитися в деяких межах, який визначається константами системи залежно від сумарного об'єму фізичної пам'яті. Наприклад, якщо фізичної пам'яті достатньо, то робочий набір процесу повинен бути в діапазоні від 50 до 345 сторінок. Маючи привілей Increase Scheduling Priority, ці значення можна змінювати за допомогою функції SetProcessWorkingSet.

Якщо виникає сторінкова помилка і робочий набір процесу не перевищив ліміту (при слабкій завантаженості системи допускається навіть перевищення ліміту), система виділяє йому ще один кадр у фізичній пам'яті. Інакше ОС намагається замінювати сторінки в робочому наборі цього процесу (локальний алгоритм заміщення).

Еволюцію робочого набору процесу можна "побачити", спостерігаючи за лічильниками Working Set та ін. в оснащенні "Продуктивність", а також за допомогою Диспетчера завдань і утиліт Pview, Pviewer і ряду інших. Важливо розуміти, що зміна робочих наборів є наслідком сторінкових порушень, які відбуваються при фактичному зверненні до сторінок пам'яті. Простого виділення і передачі пам'яті тут недостатньо.

Дослідження роботи програми, що ілюструє збільшення робочого набору процесу

Розглянемо легку модифікацію програми DemoVM із попередньої лабораторної роботи, додавши туди операцію запису одного байта на кожну сторінку переданої пам'яті (програма DemoPF.pas (PageFaults)).

Приклад 8.3. (DemoPF.pas)

```
uses windows;
var pMem:pointer=NIL;
   nPageSize:dword=4096;
```

```
    SizeCommit:dword=0;
    nPages:dword=200;
    p:^byte;
    i:dword;
begin
SizeCommit:=nPages*nPageSize;
readln;
pMem:=VirtualAlloc(NIL,SizeCommit,MEM_RESERVE or
MEM_COMMIT,PAGE_READWRITE);
if pMem = NIL then Writeln('VirtualAlloc Error');
p:=pMem;for i:=1 to nPages do begin p^:=$0FF;p:=p+nPageSize;end;
readln;
pMem:=VirtualAlloc(NIL,SizeCommit,MEM_RESERVE or
MEM_COMMIT,PAGE_READWRITE);
if pMem = NIL then Writeln('VirtualAlloc Error');
p:=pMem;for i:=1 to nPages do begin p^:=$0FF;p:=p+nPageSize;end;
readln;
pMem:=VirtualAlloc(NIL,SizeCommit,MEM_RESERVE or
MEM_COMMIT,PAGE_READWRITE);
if pMem = NIL then Writeln('VirtualAlloc Error');
p:=pMem;for i:=1 to nPages do begin p^:=$0FF;p:=p+nPageSize;end;
readln;
pMem:=VirtualAlloc(NIL,SizeCommit,MEM_RESERVE or
MEM_COMMIT,PAGE_READWRITE);
if pMem = NIL then Writeln('VirtualAlloc Error');
p:=pMem;for i:=1 to nPages do begin p^:=$0FF;p:=p+nPageSize;end;
readln;
end.
```

Нарощування об'єму переданої пам'яті і розміру робочого набору відбуватиметься по натисненні клавіші "Enter". Подивимося на поведінку лічильника "Робоча множина" для процесів DemoVM і DemoPF. Незважаючи на однаковий об'єм переданої фізичної пам'яті, розміри робочого набору сильно відрізняються. У DemoVM він залишається близьким до нуля, тоді як у процесу DemoPF йде помітний ступінчастий приріст робочого набору (див. рис. 8.5).

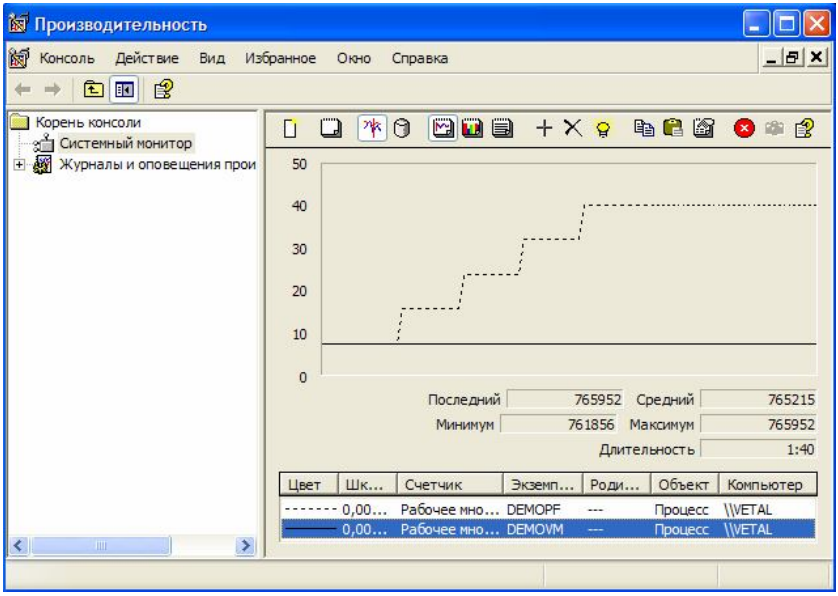


Рис. 8.5. Спостереження за змінами робочих наборів процесів

Заміщення сторінок у робочому наборі процесу – одна з найбільш відповідальних операцій. Справа в тому, що зменшення частоти page fault є одним із ключових завдань системи управління пам'яттю (наприклад, відомо, що ймовірність page fault $5*10^{-7}$ виявляється достатньою, щоб понизити продуктивність сторінкової схеми управління пам'яттю на 10 %). Розв'язання цієї задачі пов'язане з розумним вибором алгоритму заміщення сторінок. Якщо стратегія заміщення вибрана правильно, то в оперативній пам'яті залишається тільки найактуальніша інформація, яка може знадобитися в недалекому майбутньому і не потребує заміщення.

У ОС Windows використовуються алгоритми FIFO (first input first output) у багатопроцесорному варіанті і LRU – в однопроцесорному. Насправді застосовується не LRU, а його програмна реалізація NFU (not frequently used), згідно з якою сторінка характеризується не давністю, а частотою використання. Проте, згідно з документацією по ОС Windows, алгоритм, що здійснює модифікацію розміру робочого набору процесу, називається саме

називається саме LRU. Що стосується алгоритму FIFO, то попри відомі недоліки, його застосування спрощує обробку посилань на сторінку від кількох процесорів.

База даних PFN. Сторінкові сервіси

У процесі функціонування операційної системи у фізичній пам'яті розташовуються робочі набори процесів, системний робочий набір, вільні фрагменти і багато іншого. Для обліку стану фізичної пам'яті підтримується база даних PFN (page frame number). Це таблиця записів фіксованої довжини. Кількість записів у ній збігається з кількістю сторінкових кадрів.

Відомо, що підсистема віртуальної пам'яті працює продуктивно за наявності резерву вільних сторінкових кадрів. Тоді у випадку сторінкової помилки потрібна тільки одна дискова операція (читання), і вільна сторінка може бути знайдена негайно. Алгоритми, що забезпечують підтримку системи в оптимальному стані, реалізовані у складі фонових процесів (їх часто називають демонами або сервісами), які періодично "прокидаються" і інспектують стан пам'яті. Їх завдання – забезпечувати достатню кількість вільних сторінок, підтримуючи систему в стані якнайкращої продуктивності.

Формально, кожна сторінка фізичної пам'яті повинна знаходитися у складі робочого набору або входити до одного з підтримуваних базою зв'язних списків сторінок. Переміщення сторінок між списками і робочими наборами здійснюється системними потоками-демонами, що входять до складу менеджера пам'яті. Параметри настройки демонів зберігаються в розділі HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management реєстру.

Найчастіше для обслуговування помилки сторінки, відповідно до вимог захисту рівня С2, потрібна занулена сторінка, яка витягується з відповідного списку. *Список занулених* сторінок поповнюється потоком занулення сторінок (zero page thread) у фоновому режимі за рахунок списку вільних сторінок. Іноді, наприклад, для відображення файла, занулені сторінки не потрібні, і можна обійтися вільними сторінками. Якщо у робочого набору процесу відбирається сторінка, вона потрапляє в список модифікованих

сторінок або в список вільних сторінок. Підсистема запису модифікованих сторінок (modified page writer) записує їх зміст на диск, коли кількість таких сторінок перевищує встановлений ліміт. Сторінки проектованого файла можна скинути на диск явним чином (за допомогою функції FlushViewOfFile). Після запису модифікована сторінка потрапляє в список вільних сторінок.

Загальне керівництво і реалізацію загальних правил управління пам'яттю здійснює диспетчер робочих наборів (working set manager), який викликається системним потоком ядра – диспетчером настройки балансу – раз в секунду або при зменшенні об'єму вільної пам'яті нижче за порогове значення.

Експеримент. Спостереження за помилками сторінок

Кількість помилок сторінок, що генеруються процесом, можна спостерігати за допомогою лічильника "Помилок сторінки". На рис. 8.6 наведені графіки поведінки лічильників "Помилок сторінок" і "Робоча множина" для процесу DemoPF (див. програму, описану у прикладі 8.3).

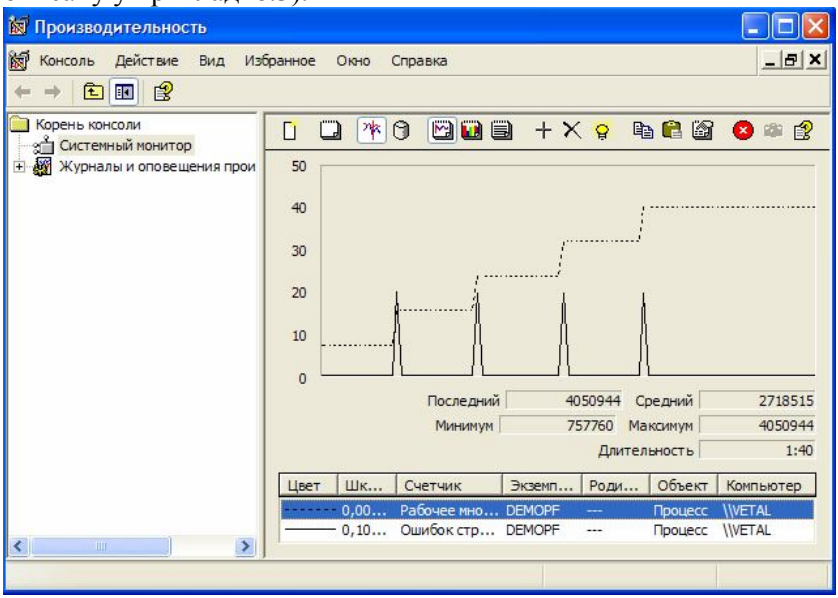


Рис. 8.6. Спостереження за розміром робочого набору процесу і кількістю сторінкових помилок

Графіки, приведені на рис. 8.6, показують, що збільшення робочого набору корелює з інтенсивністю процесів підкачки зовнішньої пам’яті.

За допомогою утиліти Pfmon.exe з ресурсів Windows можна не тільки "побачити" загальну кількість сторінкових порушень, але і визначити віртуальні адреси, звернення до яких ці порушення спровокували. На рис. 8.7 зображений фрагмент результатів роботи даної утиліти для процесу DemoPF.

```
...
SOFT: RtlFillMemoryUlong+0x10 : 0x00232000
SOFT: RtlFillMemoryUlong+0x10 : 0x00233000
SOFT: GetConsoleInputWaitHandle+0x11a :
                                GetConsoleInputWaitHandle+0x119
SOFT: FindFirstFileExA+0x285 : FindFirstFileExA+0x285
SOFT: main+0xe4 : 0x00440000
SOFT: main+0xe4 : 0x00441000
SOFT: main+0xe4 : 0x00442000
SOFT: main+0xe4 : 0x00443000
SOFT: main+0xe4 : 0x00444000
SOFT: main+0xe4 : 0x00445000
SOFT: main+0xe4 : 0x00446000
SOFT: main+0xe4 : 0x00447000
SOFT: main+0xe4 : 0x00448000
SOFT: main+0xe4 : 0x00449000
SOFT: main+0xe4 : 0x0044a000
SOFT: main+0xe4 : 0x0044b000
SOFT: main+0xe4 : 0x0044c000
SOFT: main+0xe4 : 0x0044d000
SOFT: main+0xe4 : 0x0044e000
SOFT: main+0xe4 : 0x0044f000
SOFT: main+0xe4 : 0x00450000
SOFT: main+0xe4 : 0x00451000
SOFT: main+0xe4 : 0x00452000
...
```

Рис. 8.7. Частина результатів роботи утиліти Pfmon.exe по відношенню до процесу DemoPF

Окремі аспекти функціонування менеджера пам'яті

Коректна робота менеджера пам'яті, крім принципових питань, пов'язаних із вибором абстрактної моделі віртуальної пам'яті і її апаратною підтримкою, забезпечується також множиною нюансів і дрібних деталей.

Прикладом може служити *локалізація сторінок у пам'яті*, що означає тимчасову заборону на вивантаження деяких сторінок, що зберігають буфери введення-виведення або інші важливі дані і код, наприклад, код і дані процесів реального часу.

Локалізація сторінок у пам'яті

За замовчуванням, процесу дозволяється блокувати максимум 30 сторінок пам'яті. Якщо збільшити робочу множину процесу за допомогою функції SetProcessWorkingSetSize, то, згідно з документацією, максимальна кількість сторінок, яку може блокувати процес, дорівнює мінімальному розміру його робочого набору мінус 8 сторінок.

Локалізація сторінок у пам'яті здійснюється при допомозі Win32 функції VirtualLock, а звільнення сторінок – за допомогою VirtualUnlock. Облік локалізованих сторінок ведеться в сторінковій базі PFN.

Копіювання при записі

Інший нюанс у роботі менеджера пам'яті, який можна проілюструвати на практиці, пов'язаний із реалізацією алгоритму відкладеного виділення пам'яті – копіювання при записі (copy-on-write). Це один із прикладів алгоритму відкладеної оцінки (lazy evaluation), які ускладнюють систему, але роблять її ефективнішою.

Розглянемо ситуацію, коли деяка приватна область пам'яті процесу є точною копією вже існуючого в системі фрагмента пам'яті. Наприклад, пам'ять дочірнього процесу після виклику функції fork() у Unix є копією пам'яті батьківського процесу. Інший приклад – сумісне використання динамічної бібліотеки доти, поки одна з програм не змінила її статичні дані. У таких випадках розумно не виділяти окрему область пам'яті для процесу, а відображати в його адресний простір уже існуючу. Власне виділення можна здійснити тоді, коли процес приступить до зміни

вмісту цієї області. Ця техніка називається копіюванням при записі.

Відкладене виділення пам'яті реалізоване так. Сторінки, що відображаються, позначаються прапором PAGE_WRITECOPY (доступні для читання, але, насправді, доступні для запису). Запис на таку сторінку приводить до створення її приватної копії, яка і відображається на пам'ять. Тепер можна писати на цю сторінку без ризику змінити вміст оригінальної сторінки.

Як самостійну вправу рекомендується написати програму, в якій частина сторінок файла, що відображається, була би помічена атрибутом PAGE_WRITECOPY. Запис текстового рядка в даний регіон пам'яті повинен здійснюватися після натиснення клавіші "Enter". Рекомендується здійснити дослідження роботи програми, спостерігаючи за лічильником "запис копій сторінок" при натисненні клавіші "Enter". Цікаво, що вміст початкового файла при цьому не змінюється.

Контроль процесом пам'яті іншого процесу

Ізоляція адресних просторів різних процесів є базовою парадигмою сучасних ОС і забезпечується шляхом прямого захисту пам'яті (атрибути захисту) і непрямого захисту (механізм трансляції адреси). Разом з тим, іноді виникають ситуації, коли доступ до пам'яті іншого процесу все ж таки необхідний. Зокрема, ця можливість активно використовується наладниками.

Для доступу до пам'яті процесу потрібно одержати його описувач. Найбільш природний спосіб отримання описувача – отримання описувача дочірнього процесу шляхом витягання його з параметра lProcessInformation функції CreateProcess.

Для створення регіонів у пам'яті іншого процесу можна використовувати функцію VirtualAllocEx, якій потрібно передати описувач цього процесу як параметр. Для доступу до пам'яті іншого процесу застосовуються функції ReadProcessMemory і WriteProcessMemory.

Дослідження роботи програм двох процесів із можливістю доступу одного процесу до пам'яті іншого процесу

У прикладах 8.4, 8.5 наведено дві програми, перша з яких створює консольний процес, а потім пише і зчитує дані з віртуа-

льної пам'яті цього процесу. Адреса віртуальної пам'яті, по якій дочірній консольний процес повинен записати повідомлення-відповідь, передається цьому процесу через командну строчку. Крім цього, зауважимо, що програми синхронізують свій доступ до віртуальної пам'яті, щоб коректно передати та отримати повідомлення.

Приклад 8.4

```
uses windows;
var  send:string='This is a message.';
    lpszCommandLine,buffer:string='                ';
    V:pvoid;
    D:Dword;
    hWrite,hRead:HANDLE;
    WriteEvent:string='WriteEvent';
    ReadEvent:string='ReadEvent';
    si:TSTARTUPINFO;
    pi:TPROCESSINFORMATION;
begin
V:=pvoid($00880000);
hWrite:=CreateEvent(NIL,FALSE,FALSE,(@WriteEvent)+1);
hRead:= CreateEvent(NIL,FALSE,FALSE,(@ReadEvent)+1);
ZeroMemory(@si,sizeof(si));
si.cb:=sizeof(si);
str(dword(V),lpszCommandLine);
lpszCommandLine:='CON_PROC.EXE '+lpszCommandLine;
CreateProcess(NIL,(@lpszCommandLine)+1,NIL,NIL,FALSE,
          CREATE_NEW_CONSOLE, NIL,NIL,si,pi);
V:=VirtualAllocEx(pi.hProcess,V,sizeof(send),MEM_RESERVE of
                              MEM_COMMIT,PAGE_READWRITE);
WriteProcessMemory(pi.hProcess,V,(@send)+1,sizeof(send),D);
SetEvent(hWrite);
WaitForSingleObject(hRead,INFINITE);
ReadProcessMemory(pi.hProcess,V,(@buffer)+1,sizeof(buffer),D);
writeln(buffer);
VirtualFreeEx(pi.hProcess,V,0,MEM_RELEASE);
readln;  end.
```

Приклад 8.5

```
uses windows,strings;
var  answer:string=‘This is an answer.’;
    hWrite,hRead:HANDLE;
    WriteEvent:string=‘WriteEvent’;
    ReadEvent: string=‘ReadEvent’;
    V,V_m:^char;
    adr_d:dword;
begin
hWrite:=OpenEvent(EVENT_ALL_ACCESS,FALSE,
                                                    (@WriteEvent)+1);
hRead:= OpenEvent(EVENT_ALL_ACCESS,FALSE,
                                                    (@ReadEvent)+1);
adr_d:=StrToInt(ParamStr(1));
WaitForSingleObject(hWrite,INFINITE);
V:=pvoid(adr_d);V_m:=V;
while(V^)<>chr(0)do begin
write(V^);inc(V);
end;V:=V_m;
strcopy(V,(@answer)+1);
SetEvent(hRead);
CloseHandle(hWrite);
CloseHandle(hRead);
readln;
end.
```

Висновок

Базовою операцією менеджера пам’яті є трансляція віртуальної адреси у фізичну за допомогою таблиці сторінок і асоціативної (TLB) пам’яті. У ряді випадків для реалізації пам’яті, що розділяється, інтеграції з системою введення-виведення та ін., застосовується прототипна таблиця сторінок, яка є проміжною ланкою між звичайною таблицею сторінок і фізичною пам’яттю. Для опису сторінок фізичної пам’яті підтримується база даних PFN (page frame number). Локалізацію сторінок пам’яті, контроль процесу пам’яті іншого процесу і техніку копіювання при записі мо-

жна віднести до цікавих особливостей системи управління пам'яттю ОС Windows.

Практична частина

1. Складіть програми, наведені у прикладах 8.1 – 8.5 теоретичної частини даної лабораторної роботи. Запустіть програми та переконайтеся, що вони працюють правильно (згідно з описом алгоритму їх роботи).

2. На основі наведених програм дослідіть різноманітні особливості функціонування менеджера пам'яті операційної системи Windows.

3. Оформіть звіт із виконаної лабораторної роботи, який повинен містити текст програм, демонстрацію результатів роботи, та дайте відповіді на контрольні запитання.

Контрольні запитання та завдання

1. Опишіть процес трансляції адреси в сторінковій віртуальній схемі.
2. Яким чином відбувається реалізація регіону проектованого в пам'ять файла, що розділяється між двома процесами?
3. Розкрийте поняття "робочої множини" процесу.
4. Які ви знаєте алгоритми заміщення сторінок? Опишіть їх.
5. Обґрунтуйте необхідність локалізації сторінок у пам'яті.
6. Обґрунтуйте необхідність алгоритму відкладеного виділення пам'яті.
7. Яким чином організовується контроль одним процесом пам'яті іншого процесу?

Лабораторна робота № 9
Дослідження інтерфейсу файлової системи NTFS

Теоретична частина

Вступ

У більшості комп'ютерних систем передбачені пристрої зовнішньої (вторинної) пам'яті великої місткості, на яких можна зберігати величезні об'єми даних. Щоб підвищити ефективність використання цих пристроїв, був розроблений ряд специфічних для них структур даних і алгоритмів.

Раніше прикладна програма сама розв'язувала проблеми іменування даних та їх структуризації в зовнішній пам'яті. Це ускладнювало підтримку на зовнішньому носії декількох архівів інформації, що довготривало зберігається. В даний час використовуються централізовані системи управління файлами. Система управління файлами бере на себе розподіл зовнішньої пам'яті, відображення імен файлів на адреси зовнішньої пам'яті і забезпечення доступу до даних.

Файлова система – це частина операційної системи, призначення якої полягає в тому, щоб організувати ефективну роботу з даними, що зберігаються в зовнішній пам'яті, і забезпечити користувачу зручний інтерфейс при роботі з такими даними. З погляду користувача, файл – одиниця зовнішньої пам'яті, тобто дані, записані на диск, повинні бути у складі якого-небудь файла. У ОС Windows підтримується уявлення про файл як про неструктуровану послідовність байтів. Прикладна програма має можливість зчитувати ці байти в довільному порядку. Зазвичай зберігання файла організоване на пристрої прямого доступу у вигляді набору блоків фіксованого розміру. Основне завдання підсистеми управління файлами – пов'язати символьне ім'я файла з блоками диска, які містять дані файла.

У даній лабораторній роботі основна увага буде зосереджена на NTFS – базовій файловій системі ОС Windows. Спочатку буде розглянутий інтерфейс, тобто питання структури, іменування, захисту файлів; операції над файлами; організація файлового архіву за допомогою каталогів. У наступній лабораторній роботі

будуть проаналізовані проблеми реалізації файлової системи, способи виділення дискового простору і скріплення його з ім'ям файла, забезпечення продуктивної роботи файлової системи і ряд інших питань, що цікавлять розробників системи.

Основні функції для роботи з файлами

Наочне вивчення інтерфейсу файлової системи краще почати з опису простої програми читання і запису у файл, який використовує основні (CreateFile, ReadFile і WriteFile) операції для роботи з файлами.

Дослідження роботи програми читання і запису у файл

Програма, текст якої наведений у прикладі 9.1, створює нову директорію Demo_Directory, створює в ній новий файл TestFile.tst, записує в нього деяку інформацію та закриває створений файл. Потім програма відкриває вже існуючий файл TestFile.tst, зчитує з нього повідомлення та виводить функцією MessageBox (перед цим демонструється, що область пам'яті (буфер), у який буде записано інформацію з файла, є дійсно порожньою). Далі створені файл та директорія видаляються.

Приклад 9.1

```
uses windows;
var cBuffer,cDirectoryBuffer:array[0..$400]of char;
   cTextBuffer:string='... деяка інформація, яку записано у файл
D:\\Demo_Directory\TestFile.tst ...';
   hFile:HANDLE;
   i,dwBytes:Dword;
begin
GetCurrentDirectory($400,cDirectoryBuffer);
CreateDirectory('D:\\Demo_Directory',NIL);
SetCurrentDirectory('D:\\Demo_Directory');
hFile:=CreateFile('TestFile.tst',GENERIC_WRITE,
FILE_SHARE_READ,NIL,CREATE_NEW,
FILE_ATTRIBUTE_NORMAL,0);
if(INVALID_HANDLE_VALUE=hFile)then
MessageBox(0,'Неможливо створити файл','Помилка',MB_OK);
WriteFile(hFile,cTextBuffer[1],sizeof(cTextBuffer),dwBytes,NIL);
CloseHandle(hFile);
```

```
MessageBox(0,cBuffer,’Буфер порожній:’,MB_OK);
hFile:=CreateFile(‘TestFile.tst’,GENERIC_READ,
FILE_SHARE_READ,NIL,OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL,0);
if(INVALID_HANDLE_VALUE=hFile)then
MessageBox(0,’Неможливо відкрити файл’,’Помилка’,MB_OK);
ReadFile(hFile,cBuffer,sizeof(cTextBuffer),dwBytes,NIL);
MessageBox(0,cBuffer,’Буфер заповнений:’,MB_OK);
CloseHandle(hFile);
DeleteFile(‘TestFile.tst’);
SetCurrentDirectory(cDirectoryBuffer);
RemoveDirectory(‘D:\\Demo_Directory’);
end.
```

Варіанти використання різних комбінацій параметрів функцій CreateFile, ReadFile і WriteFile детально описані у відповідній літературі. На щастя, більшість із них має цілком виразну мнеміку і не викликає ускладнень. Призначення деяких параметрів уточнюватиметься в подальшому. Важливо те, що у випадку успішного завершення функції CreateFile в системі створюється об’єкт "відкритий файл", який управляє операціями, пов’язаними з файлом, контролює сумісний доступ до файла і містить інформацію, специфічну для даного об’єкта, наприклад покажчик поточної позиції.

Після надбання деякого досвіду роботи з основними функціями введення-виведення перейдемо до розгляду найбільш важливих аспектів інтерфейсу користувача файлової системи.

Іменування файлів

Ім’я будь-якого абстрактного об’єкта – одна з його найважливіших характеристик. Коли процес створює файл, він дає йому ім’я. Після завершення процесу файл продовжує існувати і через своє ім’я може бути доступний іншим процесам. Для створення файла і привласнення йому імені в ОС Windows використовують Win32-функцію CreateFile.

Ім’я файла задається параметром lpFileName – покажчиком на рядок, що закінчується нулем. Відповідно до стандарту POSIX, ОС Windows оперує довгими (до 255 символів) іменами. Якщо

бути точнішим, максимальна довжина повного імені файла при створенні файла дорівнює MAX_PATH. Значення MAX_PATH визначено як 260, але система дозволяє подолати це обмеження і використовувати імена файлів завдовжки до 32000 символів у форматі Unicode.

У системі закладена можливість розрізняти великі і маленькі літери в назві файла (значення FILE_FLAG_POSIX_SEMANTICS параметра dwFlagsAndAttributes функції CreateFile). Проте користуватися цим прапором не рекомендується, оскільки багато додатків і пошукові програми цю можливість не враховують, тому для них даний файл може бути недоступний.

Типи файлів

ОС Windows підтримує типізацію файлів. Основні типи файлів: регулярні (звичайні) файли і директорії (довідники, каталоги).

Звичайні файли містять інформацію користувача. Директорії – системні файли, що підтримують структуру файлової системи. У каталозі міститься перелік вхідних у нього файлів і встановлюється відповідність між файлами і їх різноманітними атрибутами. Директорії будуть розглянуті нижче.

Вважається, що користувач зображує файл у вигляді лінійної послідовності байтів (притому, що реальне зберігання файла в зовнішній пам'яті організоване зовсім по-іншому). Таке уявлення виявилося дуже зручним і дозволяє використовувати абстракцію файла для організації міжпроцесних взаємодій, при роботі із зовнішніми пристроями, і т.д. Тому іноді до файлів приписують *інші об'єкти* ОС, такі як: фізичні і логічні диски, послідовні та паралельні порти, канали тощо, які створюються за допомогою тієї ж самої функції CreateFile. В цьому випадку параметр lpFileName визначає не тільки ім'я, але і тип об'єкта. Ці об'єкти розглядаються в інших роботах даного курсу.

Далі мова піде, головним чином, про звичайні *файли.*

Прикладні програми, що працюють із файлами, як правило, розпізнають тип файла по його імені відповідно до загальноприйнятих угод. Наприклад, файли з розширеннями .c, .pas – текстові файли, що зберігають програми на мовах Сі і Паскалі, а

файли з розширеннями .exe – виконувані, і т.д. Зв'язок імен з оброблювальними програмами реалізований у реєстрі.

Атрибути файлів

Окрім імені, ОС часто пов'язує з кожним файлом і іншу інформацію, наприклад дату модифікації, розмір і т.д. Ці інші характеристики файлів називаються *атрибутами*. У ОС Windows поняття атрибуту трактується ширше. Вважається, що файл – це не просто послідовність байтів, а сукупність атрибутів, і дані файла є лише одним з атрибутів – так званий неіменований потік даних. Є й інші (іменовані) потоки даних, які потрібно вказувати через двокрапку. Іменовані потоки даних можна "побачити" за допомогою таких команд, як echo і more. Наприклад, якщо виконати такі інтерактивні команди

>Echo вміст файлу > MyFile:Stream1

>more < MyFile:Stream1,

то на екрані повинні з'явитися слова "вміст файла".

Ось далеко не повний перелік атрибутів файла в NTFS:

- Стандартна інформація – біти (тільки читання, архівний), прапорів, тимчасові штампи і т.д.
- Ім'я файла. Ім'я файла зберігається в кодуванні Unicode. Імена файлів можуть повторюватися у форматі MS-DOS.
- Описувач захисту.
- Дані. Неіменований і іменовані потоки даних.
- Список атрибутів – розташування додаткових записів MFT, якщо одному запису про файл опинилося недостатньо.
- Ідентифікатор об'єкта – 64-розрядний ідентифікатор файла, унікальний для даного тому. Файл може бути відкритий не по імені, а по цьому ідентифікатору.
- Інформація про точку повторного розбору (див. наступну лабораторну роботу), яка використовується для символьних посилань і монтування пристроїв.
- Інформація про том.
- Інформація про індексування, використовувана для каталогів.
- Дані EFS (Encryption File Sysytem), використовувані для шифрування.

Ім’я файла теж є одним з атрибутів. Атрибути зберігаються у вигляді пари: <найменування атрибуту, значення атрибуту> в записі про файл у головній файловій таблиці MFT (див. наступну лабораторну роботу).

Частину атрибутів файла можна визначити при його створенні (через параметри функції CreateFile) або пізніше за допомогою SetFileAttributes, зіславшись на файл по імені. Можна також специфікувати атрибути захисту файла за допомогою параметра lpSecurityAttributes. Якщо ж значення lpSecurityAttributes рівне NIL, то відповідні атрибути файла міститимуть параметри так званого стандартного захисту.

Як приклад розглянемо просту програму, яка витягує атрибути вказаного файлу за допомогою функції GetFileAttributes.

Дослідження роботи програми отримання атрибутів файла

Приклад 9.2

```
uses windows;
var dwFileAttributes:DWORD;
begin
dwFileAttributes:=GetFileAttributes(‘pas.rar’);
if(dwFileAttributes=$0FFFFFFF)then
writeln(‘GetFileAttributes Error’) else begin
if(dwFileAttributes and FILE_ATTRIBUTE_READONLY)<>0 then
 writeln(‘The file or directory is read-only.’);
if(dwFileAttributes and FILE_ATTRIBUTE_HIDDEN)<>0 then
 writeln(‘The file or directory is hidden.’);
if(dwFileAttributes and FILE_ATTRIBUTE_SYSTEM)<>0 then
 writeln(‘The file or directory is part of, or is used exclusively by, the
operating system.’);
if(dwFileAttributes and FILE_ATTRIBUTE_DIRECTORY)<>0 then
 writeln(‘The "file or directory" is a directory.’);
if(dwFileAttributes and FILE_ATTRIBUTE_ARCHIVE)<>0 then
 writeln(‘The file or directory is an archive file or directory.’);
if(dwFileAttributes and FILE_ATTRIBUTE_NORMAL)<>0 then
 writeln(‘The file or directory has no other attributes set.’);
if(dwFileAttributes and FILE_ATTRIBUTE_TEMPORARY)<>0 then
 writeln(‘The file is being used for temporary storage.’);
```

```
if(dwFileAttributes and FILE_ATTRIBUTE_COMPRESSED)<>0
then writeln('The file or directory is compressed.');
if(dwFileAttributes and FILE_ATTRIBUTE_OFFLINE)<>0 then
 writeln('The data of the file is not immediately available.');
end;end.
```

За допомогою даної програми можна встановити характер файла pas.rar, який ви повинні попередньо створити у відповідній поточній директорії (прихований ('The file or directory is hidden.'), архів ('The file or directory is an archive file or directory.')).

Рекомендується самостійно написати програму, де застосовується функція SetFileAttributes, наприклад, встановлюється прапор "FILE_ATTRIBUTE_READONLY" (атрибут заборони запису у файл) для атрибутів вказаного файла.

Організація файлів і доступ до них. Поняття про асинхронне введення-виведення

Для зберігання файлів зазвичай використовуються пристрої прямого доступу (диски), які дозволяють звертатися безпосередньо до будь-якого блока диска. Це забезпечує довільний доступ до байтів файла, оскільки номер блока однозначно визначається *поточною* позицією усередині файла. Таким чином, файлова підсистема ОС Windows має справу з файлами, байти яких можуть бути лічені у будь-якому порядку. Такі файли називається файлами прямого доступу. Безпосереднє звернення до будь-якого байта усередині файла передбачає наявність операції позиціонування, метою якої є задання поточної позиції для зчитування або запису. Оскільки файл може мати великий розмір, покажчик поточної позиції – 64-розрядне число, для задання якого зазвичай використовуються два 32-розрядних.

Відомо, що операції введення-виведення є відносно повільними. Щоб позбавити центральний процесор від очікування виконання операції введення-виведення, в системі організована обробка асинхронних подій, зокрема переривань, для сповіщення процесора про завершення операції введення-виведення. Проте якщо на рівні ОС операції введення-виведення асинхронні, на рівні програми користувача вони ще довго залишалися синхронними і блокуючими. В результаті процес, що ініціював операцію

введення-виведення, переходив у стан очікування. Прикладом синхронного введення-виведення служить наведений вище, у прикладі 9.1, програмний фрагмент, де оператори, наступні за викликами функцій ReadFile і WriteFile, не можуть виконуватися доти, поки операція введення-виведення не завершена.

Важливим досягненням розробників ОС Windows є надання користувачу можливості здійснювати асинхронні операції введення-виведення разом із традиційними синхронними. При цьому процес, що ініціює операцію введення-виведення, не чекає її закінчення, а продовжує обчислення. У розпорядженні користувача є засоби проконтролювати завершення операції введення-виведення згодом. Асинхронне уведення-виведення дозволяє створювати ефективніші додатки за рахунок планомірного використання ресурсів і насамперед – центрального процесора.

Приклад застосування операції асинхронного читання з файла

Для того, щоб скористатися можливостями асинхронного введення-виведення, потрібно викликати функцію CreateFile зі встановленим прапором FILE_FLAG_OVERLAPPED, що входить до складу параметра dwFlagsAndAttrs, і вказати: з якої позиції здійснювати читання (запис), скільки байтів рахувати (записати) і яка подія повинна сигналізувати про те, що операція завершена. Для цього необхідно ініціювати поля структури OVERLAPPED у параметрі pOverlapped функцій ReadFile або WriteFile.

Структура OVERLAPPED

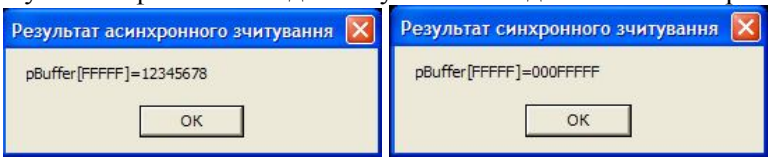
```
TOverlapped = record
Internal: DWORD;
InternalHigh: DWORD;
Offset: DWORD;
OffsetHigh: DWORD;
hEvent: THandle;    end;
```

Параметр Internal використовується для зберігання коду можливої помилки, а параметр InternalHigh – для зберігання кількості переданих байтів. Спочатку розробники Windows не планували робити їх загальнодоступними – звідси й такі імена, що не містять мнемоніки. Offset і OffsetHigh – відповідно молодші і старші

розряди поточної позиції файлу. hEvent специфікує подію, яка сигналізує про закінчення операції введення-виведення.

Дослідження роботи програми, що здійснює асинхронне читання з файла

Розглянемо програму, текст якої наведено у прикладі 9.3. Програма створює на диску файл з ім'ям „Test.dat" і заповнює його тестовим масивом даних об'ємом 4 Мбайт (1 Мбайт цілих 4-байтних чисел формату dword). Ця операція виконується звичайним синхронним чином. Далі той самий файл відкривається повторно і програма ставить до системи запит на асинхронне зчитування з нього всього масиву даних. Поки триває зчитування (при об'ємі файла 4 Мбайт на це витрачається помітний час), програма продовжує виконання, а саме – зчитує і виводить у вікно повідомлення останній елемент тестового масиву, який попередньо ініційований деяким числом: 12345678. Поява у вікні повідомлення саме цього числа говорить про те, що зчитування елемента виконується не після завершення операції введення, а паралельно з нею. Через деякий час той самий елемент масиву зчитується з файла і виводиться у вікно повідомлення повторно:



Приклад 9.3

```
uses windows,strings;
const N=$100000;
var mes:array[0..100]of char;
   i,dwCount:dword;
   ovlp:TOVERLAPPED;
   pBuffer:^dword;
   hFile:HANDLE;
   s:string;
begin      ZeroMemory(@ovlp,sizeof(ovlp));
pBuffer:=VirtualAlloc(NIL,N*4,MEM_RESERVE or
                              MEM_COMMIT,PAGE_READWRITE);
for i:=0 to N-1 do (pBuffer+i)^:=i;
```

```
hFile:=CreateFile('Test.dat',GENERIC_READ or
                  GENERIC_WRITE,0,NIL,CREATE_ALWAYS,0,0);
WriteFile(hFile,pBuffer^,N*4,dwCount,NIL);
ZeroMemory(pBuffer,N*4);
CloseHandle(hFile);
hFile:=CreateFile('Test.dat',GENERIC_READ,0,NIL,
OPEN_EXISTING,FILE_FLAG_OVERLAPPED of
                                  FILE_FLAG_NO_BUFFERING,0);
(pBuffer+i)^:=$12345678;
ReadFile(hFile,pBuffer^,N*4,dwCount,@ovlp);
s:=HEX((pBuffer+i)^);s:='pBuffer[FFFFF]='+s;
wsprintf(mes,(@s)+1);
MessageBox(0,mes,'Результат асинхронного зчитуван-
ня',MB_OK);
                          //або: writeln(s);
WaitForSingleObject(hFile,INFINITE);
ZeroMemory(@s+1,sizeof(s));
s:=HEX((pBuffer+i)^);s:='pBuffer[FFFFF]='+s;
wsprintf(mes,@s+1);
MessageBox(0,mes,'Результат синхронного зчитування',MB_OK);
                          //або: writeln(s);
VirtualFree(pBuffer,0,MEM_RELEASE);
CloseHandle(hFile);
end.
```

У програмі ініційована структура OVERLAPPED і передана функції ReadFile як параметр. Дізнатися кількість прочитаних байтів можна з ovlp.InternalHigh – компонента структури OVERLAPPED.

У програмі вибраний простий варіант синхронізації – сигналізація від об'єкта, що управляє пристроєм, в даному випадку – відкритого файла (функції WaitForSingleObject переданий описувач відкритого файла як параметр). Якщо закоментувати функцію WaitForSingleObject та виводити зчитувані числа не за допомогою MessageBox у вікна повідомлень, а безпосередньо у консольне вікно, то можна побачити, що два числа будуть мати однакові значення (12345678), тобто будуть виведені до завершення опера-

ції зчитування з файла. Існують й інші, цікавіші способи синхронізації, наприклад використання порту завершення введення-виведення (IoCompletitionPort). Детальніше можливості асинхронного введення-виведення описані у відповідній літературі.

Результат роботи даної програми практично нічим не відрізняється від звичайного синхронного читання і в такому вигляді великого сенсу не має. Проте якщо між операціями читання і синхронізації змусити програму виконувати яку-небудь корисну роботу, то ресурси комп'ютера використовуватимуться ефективніше, оскільки процесор і пристрій введення працюватимуть паралельно.

Операція позиціонування у випадку синхронного доступу до файла

Отже, у випадку асинхронного доступу позиція, починаючи з якої здійснюватиметься операція читання-запису, міститься в запиті на операцію (параметр структури OVERLAPPED). Розглянемо тепер особливості позиціонування при звичайному синхронному введенні-виведенні. В цьому випадку реалізується схема зі "збереженням стану", 64-розрядний покажчик поточної для читання-запису позиції зберігається у складі атрибутів об'єкта "відкритий файл" (його не потрібно плутати з атрибутами файла), описувач якого повертає функція CreateFile.

Поточна позиція зміщується на кінець ліченої або записаної послідовності байтів у результаті операцій читання або запису. Крім того, можна встановити поточну позицію за допомогою Win32-функції SetFilePointer. Наприклад, операція

SetFilePointer(hFile, 17, NIL, FILE_BEGIN);

встановлює покажчик поточної позиції на 17-й байт з початку файла.

Той факт, що покажчик поточної позиції є атрибутом об'єкта "відкритий файл", а не самого файла, означає, що той самий файл можна відкрити повторно з іншим описувачем. При цьому для одного і того ж файла існуватимуть два різні об'єкти з двома різними покажчиками поточних позицій. Очевидно, що зміна поточної позиції при роботі з файлом через різні об'єкти відбуватиметься незалежно.

З іншого боку, одержавши описувач відкритого файла, можна його продублювати з допомогою Win32-функції DuplicateHandle. В цьому випадку два різні описувачі посилатимуться на один і той самий об'єкт з одним і тим самим покажчиком поточної позиції.

Як самостійну вправу рекомендується написати програму, яка проілюструвала б переміщення покажчика поточної позиції в результаті операцій читання, запису і позиціонування. Необхідно також продемонструвати в програмі незалежне позиціонування для двох описувачів одного і того ж файла і залежне позиціонування для дубліката вже існуючого описувача.

Директорії. Логічна структура файлового архіву

Файлова система на диску є ієрархічною структурою, яка організована за рахунок наявності спеціальних файлів – каталогів (директорій). Каталоги мають один і той самий внутрішній табличний формат (рис. 9.1) і забезпечують багаторівневе найменування файлів.

Ім'я файла (каталогу)	*Тип файла (звичайний чи каталог)*	
Anti	К	Атрибути
Games	К	Атрибути
Autoexec.bat	З	Атрибути
Mouse.com	З	Атрибути

Рис. 9.1. Формат каталогу

Запис у каталозі про файл містить ім'я файла, деякі атрибути (довжина імені, часова мітка) і посилання на запис у головній файловій таблиці, необхідний для знаходження блоків файла.

У результаті, файлова система на диску утворює добре відому деревоподібну структуру (рис. 9.2), де немає циклів (якщо відсутні посилання і точки монтування) і шлях від кореня до файла *однозначно* визначає файл.

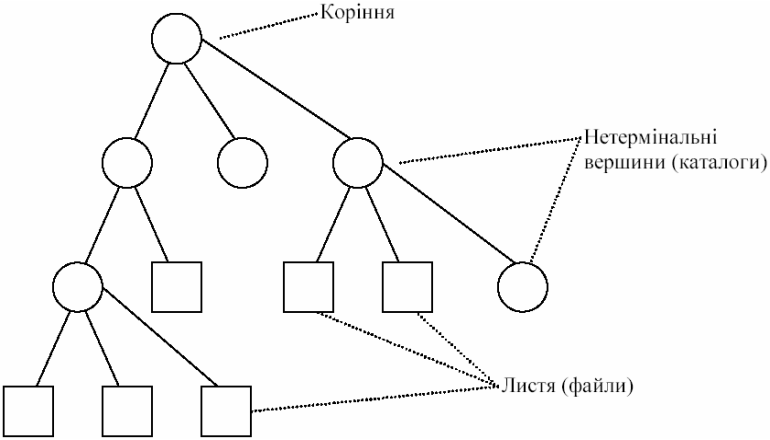


Рис. 9.2. Ієрархічна деревоподібна структура файлової системи

Оскільки імена файлів, що знаходяться в різних каталогах, можуть збігатися, унікальність імені файла на диску забезпечується додаванням до власного імені файла списку вкладених каталогів, що містять даний файл. Так утворюється добре відоме абсолютне або повне ім'я (pathname), наприклад \Games\Heroes\heroes.exe. Отже, використання деревоподібних каталогів мінімізує складність призначення унікальних імен.

Щоб мати можливість працювати з власними іменами файлів, використовують концепцію робочої або поточної директорії, яка зазвичай входить до складу атрибутів процесу, що працює з даним файлом. Тоді на файли в такій директорії можна посилатися тільки по імені. Крім того, ОС підтримує позначення '.' – для поточної директорії і '..' – для батьківської.

У системі підтримується велика кількість Win32-функцій для маніпуляції з каталогами, їх повний перелік є у відповідній літературі. Зокрема, для створення каталогів можна використовувати функцію CreateDirectory. Знов створена директорія включає записи з іменами '.' і '..', проте вважається порожньою. Для роботи з поточним каталогом можна використовувати функції GetCurrentDirectory і SetCurrentDirectory. Робота з цими функціями проста і не потребує спеціальних роз'яснень.

Дослідження роботи програми, завдання якої створити каталог на диску і зробити його поточним

Приклад 9.4

```
uses windows;
var iRet,i:integer=0;
   Buf:array[0..512]of char;
   bufSize:integer=512;
   CrDir:boolean;
begin
iRet:=GetCurrentDirectory(bufSize, Buf);
writeln;
write('iRet = ',iRet,', current directory: ');
 i:=0;while(buf[i]<>chr(0))do
begin write(buf[i]);  inc(i);end;
CrDir:=CreateDirectory('D:\tmp1', NIL);
writeln;
 if(not CrDir)then writeln('CreateDirectory error');
CrDir:=SetCurrentDirectory('D:\tmp1');
 if(not CrDir)then writeln('SetCurrentDirectory error');
iRet:=GetCurrentDirectory(bufSize,Buf);
 write('iRet = ',iRet,', current directory: ');
 i:=0;while(buf[i]<>chr(0))do
begin write(buf[i]);inc(i);end;
end.
```

Наведена програма виводить на екран назву поточного каталогу, створює каталог "tmp1" на диску "D:", робить його поточним і виводить на екран його назву як поточний каталог.

Як самостійну вправу, на основі попередньої програми рекомендується написати програму, яка створює каталог у батьківській директорії і копіює в нього який-небудь файл за допомогою функції CopyFile.

Розділи диска. Операція монтування

У ОС Windows прийнято розбивати диски на *логічні диски* (це низькорівнева операція), іноді звані розділами (partitions). Буває, що, навпаки, об'єднують декілька фізичних дисків в один логічний диск. На розділі або логічному диску зберігається кореневий

каталог даного і всі вкладені в нього каталоги, а задання шляху до файла починається з імені логічного диска або "букви" диска.

Імена логічних дисків зберігаються в каталозі "\??" простору імен об'єктів, які і здійснюють зв'язок логічного диска і реального пристрою. Вказавши букву диска, прикладна програма дістає доступ до його файлової системи.

За аналогією з Unix операційна система Windows дозволяє користувачу створити точку монтування – пов'язати який-небудь порожній каталог із каталогом логічного диска. У разі успішного завершення операції вміст цих каталогів відповідатиме один одному.

Експеримент. Монтування логічного диска за допомогою штатної утиліти mountvol

Щоб змонтувати логічний диск, потрібно виконати команду

>mountvol [<диск>:]<шлях> <ім'я тому>

Тут параметр <шлях> задає ім'я порожнього каталогу, а <ім'я тому> задається у вигляді: \\?\Volume{ код_GUID}\, де GUID – глобальний унікальний ідентифікатор.

Наприклад

>mountvol D:\tmp1 \\?\Volume\{2eca078d-5cbc-43d3-aff8-7e8511f60d0e}\

Імена глобальних унікальних ідентифікаторів і їх зв'язок із літерами диска можна дізнатися, задавши команду

>mountvol /?

Монтування також можна виконати за допомогою панелі управління дисками системної панелі управління, якщо вибрати пункт "зміна літери диска і шляху диска".

Нарешті, змонтувати диск можна програмним чином за допомогою Win32-функції SetVolumeMountPoint.

Захист файлів

Захист файлів від несанкціонованого використання ґрунтується на тому, що доступ до файла залежить від ідентифікатора користувача. Система контролю доступу припускає наявність у кожного файла дескриптора захисту, що містить список прав доступу, який формує власник файла і входить до складу атрибуту

SecurityAttributes файлу. Кожен процес має маркер доступу, який містить права користувача, що запустив процес.

Список прав доступу містить набір ідентифікаторів користувачів, що мають право на доступ до файла, і їх права відносно цього файла. Маркер доступу містить ідентифікатор власника процесу. Система контролю доступу в момент *відкриття* файла перевіряє відповідність прав власника процесу з тими, які перераховані в списку прав доступу до файла. В результаті доступ може бути дозволений або відхилений.

Під час виконання операцій читання, запису й інших перевірок прав доступу вже не проводиться.

У нових версіях NTFS дескриптори захисту всіх файлів зберігаються в окремому файлі метаданих \$Secure, який описується дев'ятим записом головної файлової таблиці тому MFT (консолідований захист).

Висновок

Отже, файл – одиниця зовнішньої пам'яті, тому зазвичай дані, записані на диск, знаходяться у складі якого-небудь файла. Файлова система розв'язує задачі іменування і типізації файлів, організації доступу до файлів, захисту, пошуку файлів і ряд інших. У системі на кожному розділі диска підтримується ієрархічна система каталогів. Для ефективного доступу до файлів можуть бути організовані асинхронне читання і запис.

Практична частина

1. Складіть програми, наведені у прикладах 9.1 – 9.4 теоретичної частини даної лабораторної роботи. Запустіть програми та переконайтеся, що вони працюють правильно (згідно з описом алгоритму їх роботи).

2. На основі наведених програм дослідіть різноманітні особливості роботи API-функцій інтерфейсу файлової системи NTFS.

3. Складіть програми, рекомендовані як самостійні вправи.

4. Оформіть звіт із виконаної лабораторної роботи, який повинен містити текст програм, демонстрацію результатів роботи, та дайте відповіді на контрольні запитання.

Контрольні запитання та завдання

1. Дайте визначення файлової системи.
2. Які основні API-функції для роботи з файлами ви знаєте?
3. Які типи файлів ви знаєте?
4. Що таке атрибути файлів?
5. Наведіть основний перелік атрибутів файлів у файловій системі NTFS.
6. В чому полягає різниця між синхронними та асинхронними операціями введення-виведення у файл?
7. Розкрийте зміст операції позиціонування. Яка API-функція пов’язана із цим поняттям?
8. Обґрунтуйте необхідність організації ієрархічної деревоподібної структури файлової системи.
9. Які основні API-функції для роботи з каталогами (директоріями) ви знаєте?
10. В чому полягає суть операції монтування?
11. Обґрунтуйте необхідність організації захисту файлів.

Лабораторна робота № 10
Дослідження особливостей реалізації файлової системи NTFS

Теоретична частина

Вступ

Типова сукупність дій користувача відносно файлової системи на диску складається з форматування диска, створення на ньому структури каталогів, заповнення їх файлами, а також виконання різноманітних дій із цими файлами. У попередній лабораторній роботі йшла мова головним чином про те, які можливості надає файлова система користувачу для виконання його завдань. Тепер перейдемо до розгляду питань, пов'язаних із реалізацією сервісів, що надаються, алгоритмами і структурами даних на диску, що дозволяють пов'язати символьне ім'я файла з даними. Крім того, файлові служби повинні розв'язувати проблеми сумісного доступу до даних, проблеми перевірки і збереження цілісності файлової системи, проблеми підвищення продуктивності і ряд інших.

Нижній рівень у системі зберігання даних – диски з рухомими головками. Для обміну з магнітним диском на рівні апаратури потрібно вказати номер циліндра, номер поверхні, номер блока на відповідній доріжці і кількість байтів, яку потрібно записати або прочитати від початку цього блока. Таким чином, диски можуть бути розбиті на блоки фіксованого розміру, і можна безпосередньо дістати доступ до будь-якого блока (організувати прямий доступ до файлів).

У традиційній багаторівневій системі побудови операційних систем із пристроями (дисками) безпосередньо взаємодіє частина ОС, звана системою введення-виведення, основу якої складають драйвери пристроїв. Завдання системи введення-виведення: приховати особливості роботи з дисками і надати в розпорядження більш високорівневого компонента ОС – файлової системи – використовуваний дисковий простір у вигляді безперервної послідовності блоків фіксованого розміру. Файлова система розташовується відповідно між системою введення-виведення і прикладною програмою.

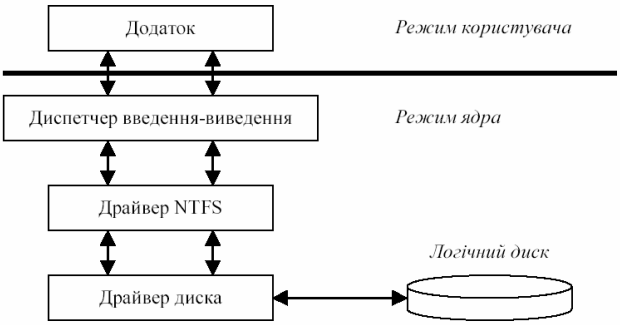


Рис. 10.1. Взаємодія додатка із системою введення-виведення та файловою системою

У ОС Windows файлова система інтегрована в систему введення-виведення (див. рис. 10.1), побудовану у вигляді набору різноманітних драйверів, і також реалізована у вигляді драйвера, наприклад драйвера NTFS або драйвера FAT. Спілкування драйверів організоване шляхом посилки так званих IRP (I/O request packet) пакетів. Функціонування системи введення-виведення детально описане у відповідній літературі. У даній лабораторній роботі мова піде про її файлову підсистему.

Кластери

Зазвичай диски розбиті на блоки (сектори) розміром – 512 байтів. Проте зручніше оперувати блоками більшого розміру – кластерами (cluster). Розмір кластера дорівнює розміру сектора, помноженому на кластерний множник (claster factor), і може бути встановлений під час операції форматування диска. За замовчуванням, це значення дорівнює 4 Кбайт і може бути змінено. Альтернативні значення розміру кластера можна, наприклад витягнути з довідкової інформації команди format.

Експеримент. Отримання інформації про потенційний розмір кластера

Нижче наведена частина результатів виведення команди "format /?":

ключ: /A: розмір

замінює розмір кластера за замовчуванням. У загальних випадках рекомендується використовувати розміри кластера за замовчуванням.

NTFS підтримує розміри 512, 1024, 2048, 4096, 8192, 16КБ, 32КБ, 64К. FAT підтримує розміри 512, 1024, 2048, 4096, 8192, 16КБ, 32КБ, 64КБ (128КБ, 256КБ для розміру сектора більше 512 Байт).

FAT32 підтримує розміри 512, 1024, 2048, 4096, 8192, 16КБ, 32КБ, 64КБ (128КБ, 256КБ для розміру сектора більше 512 Байт).

Файлові системи FAT і FAT32 накладають такі обмеження на кількість кластерів тому:

FAT: кількість кластерів <= 65526

FAT32: 65526 < кількість кластерів < 4177918

Виконання команди Format буде негайно перервано, якщо буде виявлене порушення вказаних вище обмежень, використовуючи вказаний розмір кластерів.

Стиснення томів NTFS не підтримується для розмірів кластерів більше 4096 Байт.

Розмір кластера, які надалі також називатимуться блоками диска, відіграє важливу роль. Невеликий розмір блока призводитиме до того, що кожен файл міститиме багато блоків і читатися поволі. Великі блоки забезпечують вищу швидкість обміну з диском, але через внутрішню фрагментацію (кожен файл займає цілу кількість блоків, і в середньому половина останнього блока пропадає) знижується відсоток корисного дискового простору. Спеціально проведені дослідження показали, що оптимальним є компромісний розмір блока, який лежить у діапазоні від 1-го до 8 Кб.

Система розрізняє кластери диска (volume cluster) і кластери диска, що належать файлу (logical cluster). Для них підтримується різна нумерація, відповідно VCN і LCN.

Дослідження роботи програми, яка отримує інформацію про диски

Наведена у прикладі 10.1 програма за допомогою API-функцій GetLogicalDrives, GetVolumeInformation та GetDiskFreeSpace видає інформацію про ті диски, які встановлені на комп'ютері. Під час запуску програма визначає кількість дисків, установлених на

комп'ютері, і включає вказівку на кожен диск у меню „Диски". Після вибору якого-небудь з елементів меню, який відповідає одному з дисків, на екрані виникає вікно повідомлень з інформацією про даний диск.

Приклад 10.1

```
uses windows,messages;
{$r menu.res}
const DISK_MENU = 10;
     IDM_EXIT = 100;
type _FLAGDISK =  packed record
    dwFlag:Dword;
    cDisk:array[0..4]of char;end;
type argList =  packed record
    cVolume:^char;
    cVNB:^char;
    dwVSN:DWORD;
    dwMCL:DWORD;
    dwFSF:DWORD;
    cFSNB:^char;
    end;
type argList2 =  packed record
    dwSPC:DWORD;
    dwBPS:DWORD;
    dwNOFC:DWORD;
    dwTNOC:DWORD;
    end;
var FlagDisk:array[0..25]of _FLAGDISK;
AllD:string='A:\\B:\\C:\\D:\\E:\\F:\\G:\\H:\\I:\\J:\\K:\\L:\\M:\\N:\\O:\\P:\\
                                    Q:\\R:\\S:\\T:\\U:\\V:\\W:\\X:\\Y:\\Z:\\';
   dwDisks,i,j,k,m:Dword;
   p:argList;p2:argList2;
procedure ViewDiskInfo(wld:Word);
var cBuffer,cBuffer2,cVolumeNameBuffer,cFileSystemNameBuffer:
                                                    array[0..$400]of char;
```

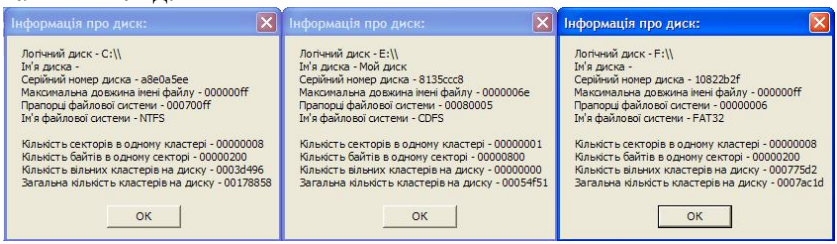
```
begin
if GetVolumeInformation(FlagDisk[wld].cDisk,cVolumeNameBuffer,
$80,@p.dwVSN,p.dwMCL,p.dwFSF,cFileSystemNameBuffer,$80)
then begin
p.cVolume:=@FlagDisk[wld].cDisk;
p.cVNB:=@cVolumeNameBuffer;
p.cFSNB:=@cFileSystemNameBuffer;
wvsprintf(cBuffer,'Логічний диск - %s'+#10+'Ім''я диска -
%s'+#10+ 'Серійний номер диска - %08x'+#10+'Максимальна
довжина імені файла - %08x'+#10+ 'Прапорці файлової системи -
%08x'+#10+'Ім''я файлової системи - %s',@p);
GetDiskFreeSpace(FlagDisk[wld].cDisk,p2.dwSPC,
                                 p2.dwBPS,p2.dwNOFC,p2.dwTNOC);
wvsprintf(cBuffer2,#10+#10+'Кількість секторів в одному кластері
- %08x'+#10+'Кількість байтів в одному секторі - %08x'+#10+
'Кількість вільних кластерів на диску - %08x'+#10+'Загальна
кількість кластерів на диску - %08x',@p2);
i:=0;while(cBuffer[i]<>#0)do inc(i);
j:=0;repeat cBuffer[i]:=cBuffer2[j];inc(i);inc(j);until(cBuffer2[j-1]=#0);
MessageBox(0,cBuffer,'Інформація про диск:',MB_OK);
    end else MessageBox(0,'Можливо, немає дискети або оптичного
диска в дисководі','Помилка',MB_OK);
end;
function WindowProc conv arg_stdcall
(Window:HWND;Mess:UINT;Wp:WParam;Lp:LParam):LRESULT;
begin
  case Mess of
      WM_CREATE:
      begin
      dwDisks:=GetLogicalDrives;
      j:=0;for i:=0 to 25 do
      if(dwDisks and FlagDisk[i].dwFlag)<>0 then begin
InsertMenu(GetSubMenu(GetMenu(Window),0),j,MF_STRING or
MF_BYPOSITION,IDM_EXIT+i+1,FlagDisk[i].cDisk);inc(j);end;
      end;
      WM_COMMAND:
```

```
      begin
       case LoWord(Wp) of
         IDM_EXIT:SendMessage(Window,WM_DESTROY,0,0);
       else ViewDiskInfo(LoWord(Wp)-IDM_EXIT-1);
       end;
      end;
      WM_DESTROY:PostQuitMessage(0);
  else Result:=DefWindowProc(Window, Mess, Wp, Lp);
  end;
end;
var wc:TWndClass;wnd:HWnd;Msg:TMsg;
begin
j:=1;k:=1;for i:=0 to 25 do begin
for m:=0 to 3 do begin FlagDisk[i].cDisk[m]:=AllD[j];inc(j);end;
FlagDisk[i].dwFlag:=k;k:=k shl 1;end;
  FillChar(wc, SizeOf(wc), 0);
  with wc do begin
    style:=CS_HREDRAW + CS_VREDRAW;
    lpfnWndProc := @WindowProc;
    cbClsExtra := 0;
    cbWndExtra := 0;
    hInstance := System.hInstance;
    hIcon := LoadIcon(THandle(NIL), IDI_APPLICATION);
    hCursor := LoadCursor(THandle(NIL), IDC_ARROW);
    hbrBackGround := GetStockObject(WHITE_BRUSH);
    lpszMenuName := MAKEINTRESOURCE(DISK_MENU);
    lpszClassName := 'Лабораторна робота № 10';          end;
  if RegisterClass(wc) = 0 then Exit;
    wnd := CreateWindow(wc.lpszClassName,'Програма дослідження
файлових систем Windows', WS_OVERLAPPEDWINDOW,
                                  200,200,450,180,0,0,HInstance,nil);
  ShowWindow(wnd, SW_RESTORE);
  UpdateWindow(wnd);
  while GetMessage(Msg,0,0,0) do  begin
    TranslateMessage(Msg);
    DispatchMessage(Msg);  end;  end.
```

Для своєї роботи програма використовує такий файл ресурсів:

```
10 MENU DISCARDABLE
  BEGIN
  POPUP "&Диски"
      BEGIN
      MENUITEM SEPARATOR
      MENUITEM "&Вихід",100
      END
  END
```

Вікна повідомлень, які видають інформацію про диски, мають такий вигляд:



Структури даних, необхідні для опису файлової системи на диску

Основна функція файлової системи – зв'язок символьного імені файла і блоків диска, що належать файла, – реалізується за допомогою посилання із запису каталогу про даний файл на запис у таблиці, формат якої визначається типом файлової системи на даному дискові.

Наприклад, у файловій системі FAT, однієї з файлових систем, підтримуваних ОС Windows, є таблиця відображення файлів (file allocation table), яка підтримує зв'язний список блоків для кожного файла. Таблиця індексована по номерах блоків. Запис у каталозі вказує на рядок у таблиці, що містить перший блок файла, а далі по таблиці можна знайти решту блоків даного файла. Принцип роботи файлової системи FAT детальніше описаний у відповідній літературі.

Розглянемо будову базової файлової системи ОС Windows – NTFS.

Головна файлова таблиця MFT

У файловій системі NTFS запис про файл у каталозі зіставляється із записом про файл у головній файловій таблиці диска – MFT (master file table), яка містить інформацію про розташування даних файла.

MFT – головна структура даних на диску, яка являє собою звичайний файл, що містить до 248 записів розміром 1 Кб кожна (див. рис. 10.2). Кожному файлу або каталогу відповідає один запис. Записи 0-15 зарезервовані для службових файлів, а записи, починаючи з 16-ї, призначені для файлів користувачів. Для великих файлів потрібно декілька записів, перша з яких називається базовою. Таблиця MFT може розташовуватися в будь-якому місці диска.

18	**...**
17	Другий файл користувачів
16	Перший файл користувачів
15 14 13 12	Зарезервовані
11	Розширення метаданих, квоти
10	Перетворення регістра
9	Файл описувачів захисту
8	Список поганих кластерів
7	Завантажувальний сектор
6 5	Бітовий масив обліку зайнятих кластерів
4	Кореневий каталог
3	Таблиця визначення атрибутів
2	Файл тому
1	Файл журналу для відновлення
0	Дзеркальна копія MFT

Рис. 10.2. Записи MFT

До складу кожного запису входить заголовок і послідовність пар <заголовок атрибуту, значення>. Якщо атрибут цілком поміщається в записі MFT, він вважається резидентним. Інакше атрибут поміщається в окремі блоки диска, а в заголовку атрибуту зберігається інформація про його місцезнаходження. Завжди резидентними є атрибути: "ім'я файла", "стандартна інформація", а такі атрибути, як "потік даних файла", "індекс" великого каталогу, зазвичай нерезидентні, хоча для файлів розміром кілька сот байтів потік даних може бути резидентним атрибутом, оскільки цілком поміщається в записі MFT.

Частіше дані файла все ж таки не поміщаються в записі MFT. Розглянемо даний варіант дещо докладніше. В цьому випадку вслід за заголовком у записі розміщується список дискових блоків файла (рис. 10.3).

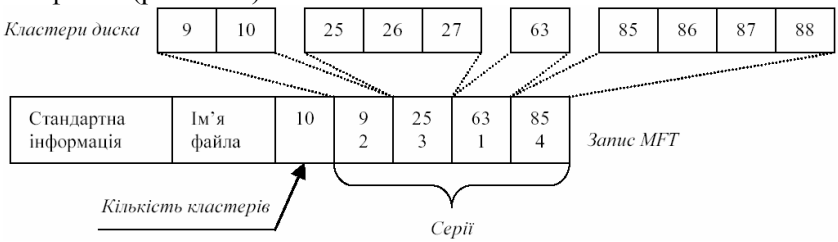


Рис. 10.3. Запис MFT для 10-блокового файла, що складається з чотирьох фрагментів (серій)

Нагадаємо, що виконується завдання приведення у відповідність номера блока у файлі (LCN) номера блока на диску (VCN). Для цього блоки диска подаються у вигляді сукупності серій, кожна з яких є безперервною послідовністю блоків. Наприклад, на рис. 10.3 показано відображення 10-блокового файла, блоки якого розміщуються в 9, 10, 25, 26, 27, 63, 85, 86, 87 і 88-му блоках диска. Схема досить ефективна, особливо для не дуже фрагментованих файлів, наприклад, безперервний файл незалежно від розміру описується всього однією серією.

Для сильнофрагментованих файлів потрібно багато серій і кілька MFT записів. Перший запис про файл містить список решти

записів. Якщо цей список великий, то він є нерезидентним атрибутом і розміщується в окремому файлі.

Номери дискових кластерів файлів можна дізнатися за допомогою утиліти nfi.exe (NTFS File Sectors Information Util), що входить до складу ресурсів Windows.

Експеримент. Перегляд кластерів, що належать файлу, за допомогою утиліти nfi.exe

```
\TMP\Nfi\exp.h
   $STANDARD_INFORMATION (resident)
   $FILE_NAME (resident)
   $DATA (nonresident)
      logical sectors 471790-471794 (0x732ee-0x732f2)
File 33
\TMP\Nfi\h.h
   $STANDARD_INFORMATION (resident)
   $FILE_NAME (resident)
   $DATA (nonresident)
      logical sectors 471798-471809 (0x732f6-0x73301)
```

Тут наведена інформація про файли \tmp\nfi\exp.h і \tmp\nfi\h.h, яку видає утиліта nfi. Найцікавіше розташування на диску нерезидентного атрибуту потоку даних, дискові номери кластерів якого в даному випадку позначаються як logical sectors.

Управління вільним і зайнятим дисковим простором

У системі Windows облік вільних і зайнятих дискових блоків ведеться за допомогою бітового вектора (bit map або bit vector), наприклад, 00111100111100011000001, де кожен блок представлений одним бітом, що набуває значення 0 або 1, залежно від того, зайнятий він чи вільний. У файловій системі NTFS бітовий масив сам є файлом. Його атрибути і дискові адреси зберігаються в шостому записі таблиці MFT.

Реалізація директорій

Як уже мовилося, директорія або каталог – це файл, що має вигляд таблиці і зберігає список вхідних у нього файлів або каталогів. Основне завдання файлів-директорій – підтримка ієрархічної деревоподібної структури файлової системи.

Як і будь-якому файлу, каталогу відповідає запис у таблиці MFT. Цей запис включає сукупність записів про файли, що входять у даний каталог, і індексована так, щоб забезпечити ефективний пошук імені файла. Кожен запис про файл включає його ім'я, мітку часу, розмір і посилання на MFT-запис для даного файла. Все це дозволяє пошуковим програмам швидко одержувати основну інформацію про файл із запису в каталозі без звернення до MFT-запису самого файла. Запис MFT для невеликого каталогу, де записи про файли є резидентним атрибутом, показаний на рис. 10.4.

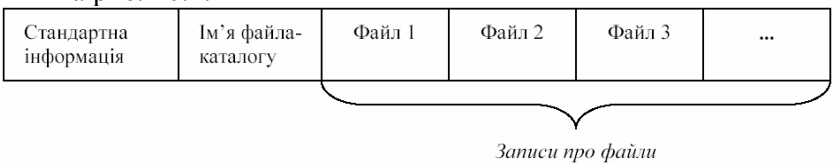


Рис. 10.4. MFT запис для невеликого каталогу

Для великих каталогів сукупність записів про файли не поміщається в MFT-запис каталогу. Вона є нерезидентним атрибутом і організована у вигляді B+ дерева, що забезпечує швидкий пошук імені файла в алфавітному порядку. MFT-запис каталогу містить корінь цього дерева, а його гілки розміщуються в окремих блоках диска.

Пошук файла по імені

Пошук файла на диску – стандартне завдання, яке доводиться виконувати будь-якому працюючому з файлами додатку вже на етапі відкриття файла. Пошукові програми використовують для цих цілей спеціалізовані API-функції пошуку файлів.

При пошуку файла спочатку за допомогою механізму символьних посилань у просторі імен об'єктів виконується завдання трансляції імені диска "в стилі DOS" або літери диска у внутрішні імена пристроїв Windows. Для цього бібліотечний виклик, що містить ім'я файла як параметр, передається бібліотеці kernel32.dll, і перед ім'ям поміщається назва каталога іменованих ресурсів "\??\" у просторі імен менеджера об'єктів. В результаті "F:\tmp\MyFile.txt" перетвориться в "\??\F:\tmp\MyFile.txt". Далі в каталозі \??\ шукається символьне ім'я "F:", яке є посиланням на

об’єкт-розділ жорсткого диска, наприклад "\Device\Harddisk\Volume5". Далі знаходиться таблиця MFT цього розділу, потім здійснюється навігація по каталогах і знаходиться потрібний файл (див. рис. 10.5).

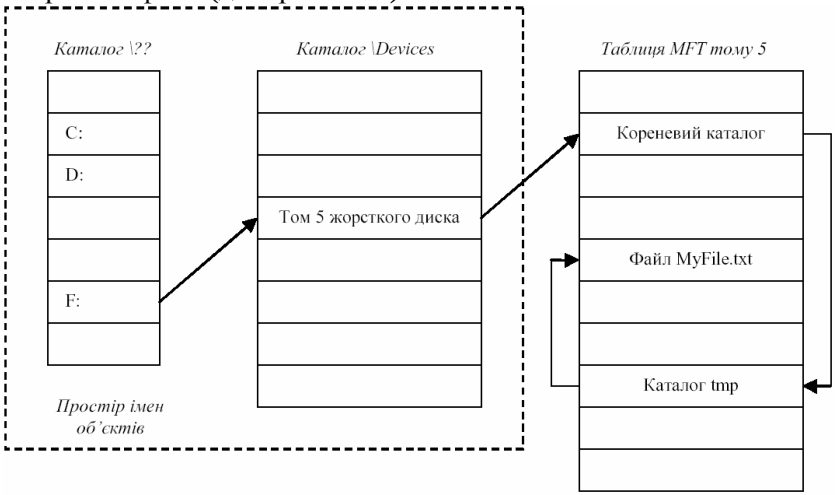


Рис. 10.5. Процес пошуку файла по імені

Для пошуку файлів у каталозі застосовуються функції FindFirstFile і FindNextFile.

Дослідження роботи програми, що здійснює пошук файлів у каталозі

Наведена програма виводить список файлів у каталозі із заданим шаблоном пошуку. Інформація про знайдені файли міститься в структурі WIN32_FIND_DATA, яка є одним із параметрів функцій FindFirstFile і FindNextFile. Вхідним параметром функції FindFirstFile служить шаблон пошуку, а вихідним – дескриптор пошуку, який є вхідним параметром функції FindNextFile і зберігає інформацію про поточний стан пошуку (аналог покажчика поточної позиції). Для закриття описувача в даному випадку застосовується функція FindClose (а не CloseHandle як завжди).

Окрім виведення назв файлів, виводиться також час останньої зміни у файлі та час створення файла. Форматування часу відбу-

вається двома різними методами з використанням відповідно API-функції FileTimeToDosDateTime та FileTimeToSystemTime.

Приклад 10.2

```
uses windows;
var hFindFile: HANDLE;
   fd:TWin32FindData;
   FatDate,FatTime,day,month,year,hour,min,sec:word;
   ft:TFileTime;
   st:TSystemTime;
procedure FileTimeToDosTime;
begin
day:=FatDate and $1f;FatDate:=FatDate shr 5;
month:=FatDate and $f;FatDate:=FatDate shr 4;
year:=(FatDate and $7f)+1980;
sec:=2*(FatTime and $1f);FatTime:=FatTime shr 5;
min:=FatTime and $3f;FatTime:=FatTime shr 6;
hour:=FatTime and $1f;
end;
begin
hFindFile:=FindFirstFile('C:\\TMTPL\*',fd);
if(hFindFile=INVALID_HANDLE_VALUE)then
  writeln('Find first file failed.')else begin
FileTimeToLocalFileTime(fd.ftLastWriteTime,ft);
FileTimeToDosDateTime(ft,FatDate,FatTime);
FileTimeToDosTime;
writeln('The first file name: ',fd.cFileName,', Last Write Time: '
                              ,day,'/',month,'/',year,' ',hour,':',min,':',sec);
FileTimeToLocalFileTime(fd.ftCreationTime,ft);
FileTimeToSystemTime(ft,st);
writeln('                      Creation Time: '
                                    ,st.wDay,'/',st.wMonth,'/',st.wYear,'
                                ',st.wHour,':',st.wMinute,':',st.wSecond);
while(FindNextFile(hFindFile,fd))do begin
    FileTimeToLocalFileTime(fd.ftLastWriteTime,ft);
    FileTimeToDosDateTime(ft,FatDate,FatTime);
    FileTimeToDosTime;
```

```
    writeln('The next file name: ',fd.cFileName,', Last Write Time: '
                              ,day,'/',month,'/',year,' ',hour,':',min,':',sec);
FileTimeToLocalFileTime(fd.ftCreationTime,ft);
FileTimeToSystemTime(ft,st);
writeln('                  Creation Time: '
                                    ,st.wDay,'/',st.wMonth,'/',st.wYear,'
                              ',st.wHour,':',st.wMinute,':',st.wSecond);end;
FindClose(hFindFile);
end; end.
```

Точки повторного аналізу. Монтування дисків. Утворення посилань

Сучасні операційні системи зазвичай надають у розпорядження користувача типові можливості для монтування файлових систем і утворення жорстких і символічних зв'язків. Ця функціональність в ОС Windows реалізується за допомогою механізму, званого точками повторного аналізу. Якщо файл помічений як точка повторного аналізу, то у складі його атрибутів наявний прапорець FILE_ATTRIBUTE_REPARSE_POINT. Зазвичай із цим файлом асоційований блок даних, які можуть бути лічені й інтерпретовані додатком, що створив цю точку, або драйвером ОС.

Монтування файлових систем

Операція монтування файлової системи, що зберігається на розділі диска, забезпечує їй зв'язок з уже існуючою ієрархією файлових систем і робить її файли доступними для процесів. Техніка монтування описана в попередній лабораторній роботі. Монтування базових дисків здійснюється автоматично при першому зверненні до диска. Це робить диспетчер монтування (Mountmgr.sys). Відомості про вмонтовувані диски є в реєстрі в розділі HKLM\SYSTEM\MountedDevices.

Створення точок монтування (mount points) – скріплення каталогу NTFS реалізоване за допомогою точок повторного аналізу. Програми, що здійснюють навігацію по каталогах, повинні, виявивши точку повторного аналізу (в даному випадку – точку монтування), вдатися до допомоги коду, що спрямовує процес подальшої навігації на новий том. Програми, що надають інформацію

про сумарну кількість файлів на диску або в каталозі, також повинні враховувати наявність точок монтування.

Точки монтування в системі можна знайти з допомогою Win32-функцій FindFirstVolumeMountPoint і FindNextVolumeMountPoint.

Створення зв'язків

Скріплення файлів – техніка, запозичена з Unix, – утворення для файла або каталогу кількох батьківських каталогів (див. рис. 10.6).

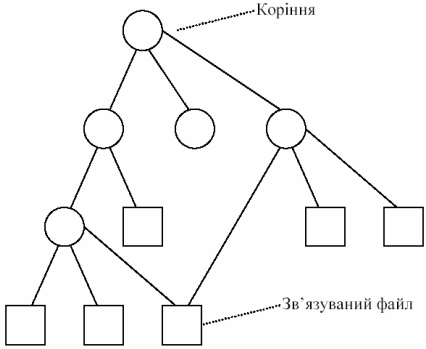


Рис. 10.6. Утворення зв'язків у файловій системі

З'єднання між директорією і файлом, що розділяється, називається "зв'язком" або "посиланням" (link). Дерево файлової системи перетворюється в циклічний граф.

ОС Windows (як і Unix) підтримує два види зв'язків – жорсткі (hard link) і символічні (symbolic link). У випадку жорсткого зв'язку запис про файл з'являється в новому каталозі, а MFT-запис цього файла включає лічильник кількості посилань на даний файл. Видалення файла приводить до зменшення лічильника на 1, і реальне видалення і звільнення його блоків відбувається, коли значення лічильника дорівнює 0.

Символічне зв'язування – створення нового файла, який містить шлях до зв'язуваного файла. Зазвичай у системі створюється каталог, який зв'язується з уже існуючим каталогом. Цей метод зручний для "підйому" (зменшення ступеня вкладеності) каталогів. Видалення символічного зв'язку на зв'язуваний файл ніяк не

впливає. Видалення зв'язуваного файла робить символічний зв'язок недійсним.

Жорсткі зв'язки створюються викликом Win32-функції CreateHardLink. Штатної утиліти, що підтримує жорсткі зв'язки, немає, хоча до складу ресурсів Windows для цих цілей включена POSIX утиліта ln.

Символічний зв'язок (іноді його називають точкою з'єднання, junction) можна утворити за допомогою вхідної до складу ресурсів Windows утиліти linkd.exe або за допомогою вільно поширюваної утиліти junction.exe.

Той факт, що операції монтування і скріплення можуть перетворити ієрархічне дерево файлової системи у циклічний граф, робить роботу з нею складнішою. Оскільки тепер до файла існує декілька шляхів, програма пошуку файла може знайти його на диску кілька разів. Простий практичний розв'язок даної проблеми – обмежити кількість директорій при пошуку. З цією метою при пошуку файлів Windows Explorer зупиняє рекурсію після досягнення 32-го рівня вкладеності або при перевищенні довжини шляху в 256 символів. Це залежить від того, яка подія наступить раніше.

Повне усунення циклів при пошуку – досить трудомістка процедура, що виконується спеціальними утилітами і пов'язана з багатократним трасуванням директорій файлової системи.

Сумісний доступ до файла

Користувачі часто потребують розділення файлів і сумісного доступу до них.

Як уже було сказано, операція відкриття файла має наслідком створення об'єкта "відкритий файл". Специфіка об'єкта "відкритий файл" полягає в тому, що він містить лише унікальні дані (наприклад, покажчик поточної позиції), тоді як власне файл – спільно використовувані дані. Тому, якщо двічі здійснити операцію відкриття одного і того ж файла, то система створить два об'єкти "файл". Дана ситуація проілюстрована рис. 10.7 для випадку одночасного відкриття файла потоками різних процесів.

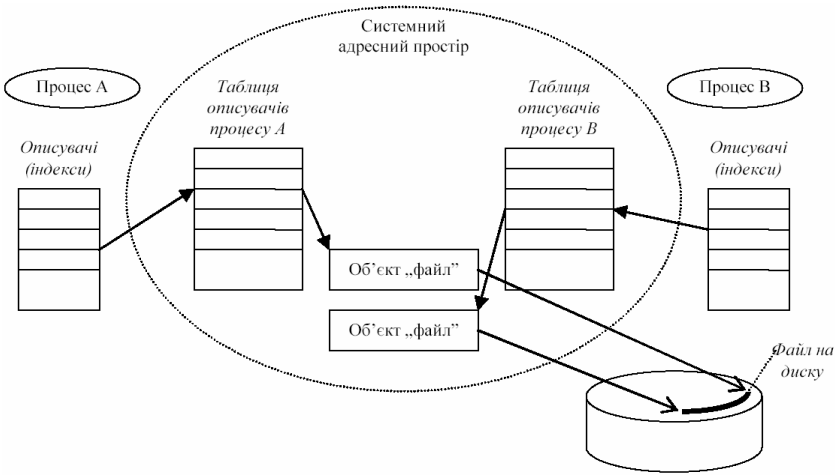


Рис. 10.7. Організація сумісного доступу до файла

Очевидно, що потоки повинні синхронізувати доступ до спільно використовуваних файлів або каталогів, щоб отримати передбачений результат. Між двома операціями read одного потоку інший потік може модифікувати дані, що для багатьох додатків неприйнятно. ОС Windows пропонує стандартне рішення цього питання на рівні користувача – надати можливість одному з потоків захопити частину файла між двома записами для монопольного доступу. Для цього використовуються Win32-функції LockFile і UnlockFile.

Дослідження механізму захоплення частини файла для монопольного доступу

Програма, текст якої наведено у прикладі 10.3, демонструє неможливість здійснення операції запису одним процесом у файл, заблокований іншим процесом за допомогою функції LockFile.

Приклад 10.3

```
uses windows;
var cTextBuffer:string='... деяка інформація, яку записано у файл
demo_file.txt ...';
   V:pvoid;
   hDup,hFile:HANDLE;
```

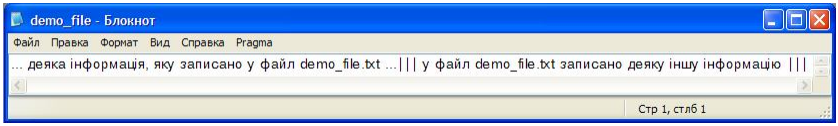
```
   D,i,dwBytes,ID:DWORD;
   IpszAppName:string='UNLOCK_F.EXE';
   si:TSTARTUPINFO;
   piApp:TPROCESSINFORMATION;
   lpszCommandLine:string;
begin
ID:=GetCurrentProcessId;
OpenProcess(PROCESS_DUP_HANDLE,true,ID);
hFile:=CreateFile('demo_file.txt',GENERIC_READ or
GENERIC_WRITE,FILE_SHARE_READ or
FILE_SHARE_WRITE,NIL,CREATE_ALWAYS,
                                    FILE_ATTRIBUTE_NORMAL,0);
if(INVALID_HANDLE_VALUE=hFile)then
MessageBox(0,'Неможливо відкрити файл','Помилка',MB_OK);
ZeroMemory(@si,sizeof(si));
si.cb:=sizeof(si);
V:=pvoid($00880000);
str(dword(V),lpszCommandLine);
lpszCommandLine:='UNLOCK_F.EXE '+lpszCommandLine;
CreateProcess(NIL,@lpszCommandLine[1],NIL,NIL,FALSE,
                        CREATE_NEW_CONSOLE,NIL,NIL,si,piApp);
DuplicateHandle(GetCurrentProcess,hFile,piApp.hProcess,@hDup,0,
                                    true,DUPLICATE_SAME_ACCESS);
V:=VirtualAllocEx(piApp.hProcess,V,sizeof(hDup),MEM_RESERVE
                             or MEM_COMMIT,PAGE_READWRITE);
WriteProcessMemory(piApp.hProcess,V,@hDup,sizeof(hDup),D);
SetFilePointer(hFile,0,nil,FILE_BEGIN);
if not LockFile(hFile,0,0,length(cTextBuffer),0)then
MessageBox(0,'Неможливо заблокувати задану область
                                                файла','Помилка',MB_OK);
for i:=1 to length(cTextBuffer)do begin sleep(10);
WriteFile(hFile,cTextBuffer[i],1,dwBytes,NIL);end;
     UnlockFile(hFile,0,0,length(cTextBuffer),0);
readln;
CloseHandle(piApp.hThread);
CloseHandle(piApp.hProcess);
```

```
CloseHandle(hFile);
VirtualFreeEx(piApp.hProcess,V,0,MEM_RELEASE);
end.
```

Текст програми дочірнього процесу UNLOCK_F.PAS:

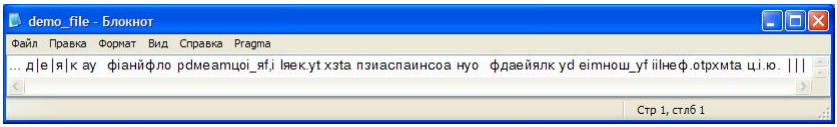
```
uses windows,strings;
var cTextBuffer:string='||| у файл demo_file.txt записано деяку іншу
інформацію |||';
   hFile:HANDLE;
   adr_d,i,dwBytes:DWORD;
   V:^dword;
begin
sleep(10);
adr_d:=StrToInt(ParamStr(1));
V:=pvoid(adr_d);
hFile:=HANDLE(V^);
repeat WriteFile(hFile,'?',1,dwBytes,NIL);
until dwBytes=1;
SetFilePointer(hFile,-1,nil,FILE_CURRENT);
for i:=1 to length(cTextBuffer)do begin sleep(10);
WriteFile(hFile,cTextBuffer[i],1,dwBytes,NIL);end;
CloseHandle(hFile);
end.
```

У результаті запуску першої програми буде коректний послідовний запис спочатку батьківським процесом деякого повідомлення у файл (при цьому блокується запис із боку будь-яких інших процесів), а потім відбувається запис у файл дочірнім процесом іншого повідомлення:



Якщо у тексті програми прикладу 10.3 закоментувати рядки, де використовуються API-функції LockFile і UnlockFile, то обидва

процеси будуть записувати у файл власні повідомлення у змішаному порядку, в результаті чого вміст файла буде некоректним та непередбачуваним:



У прикладі 10.3 використано механізм дублювання дескрипторів, зокрема дескриптора файла, за допомогою API-функції DuplicateHandle. Дескриптор відкритого файла передається через розподілену пам'ять, адреса якої передається через командний рядок при створенні нового процесу (аналогічно, як це зроблено в одній із попередніх лабораторних робіт). Таким чином, дочірній процес заново не створює дескриптор.

Продуктивність файлової системи

Ефективність роботи – одне з найважливіших завдань підсистеми управління файлами.

Кешування

Найприродніший спосіб підвищити продуктивність – мінімізувати кількість звернень до диска за рахунок *кешування*. У загальному випадку кешування – використання швидкого пристрою для оптимізації роботи повільного пристрою. В даному випадку частина блоків файла, з якими активно взаємодіє додаток, розміщується в буфері оперативної пам'яті. Періодично проводиться синхронізація вмісту кешу і диска. Можливість використання кешу в операційних системах зумовлена властивістю локальності – об'єм ключової інформації в кожен момент часу відносно невеликий.

Пристрій кешу ОС Windows відрізняється від традиційного. У традиційній реалізації кеш – буфер в оперативній пам'яті, що містить ряд блоків диска і розташований між файловою системою і системою введення-виведення. Якщо є запит на читання (запис) у файл, файлова система обчислює номер блока у файлі і номер відповідного йому блока диска. Перед читанням/записом блока диска проводиться перевірка на предмет наявності цього блока в

кеші. Якщо блок у кеші є, то запит задовольняється з кешу, інакше запитаний блок зчитується в кеш з диска.

У ОС Windows кеш працює на вищому рівні, ніж файлова система (див. рис. 10.8).

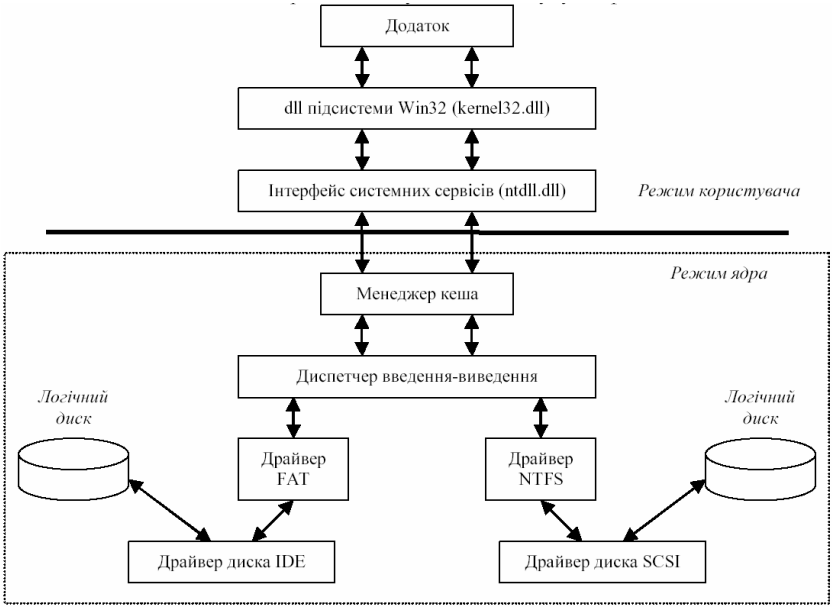


Рис. 10.8. Місце менеджера кешу в системі введення-виведення

За допомогою техніки файлів, що відображаються в пам'ять, частина зчитуваного (записуваного) файла проектується в 256-кілобайтний буфер кешу (див. рис. 10.9).

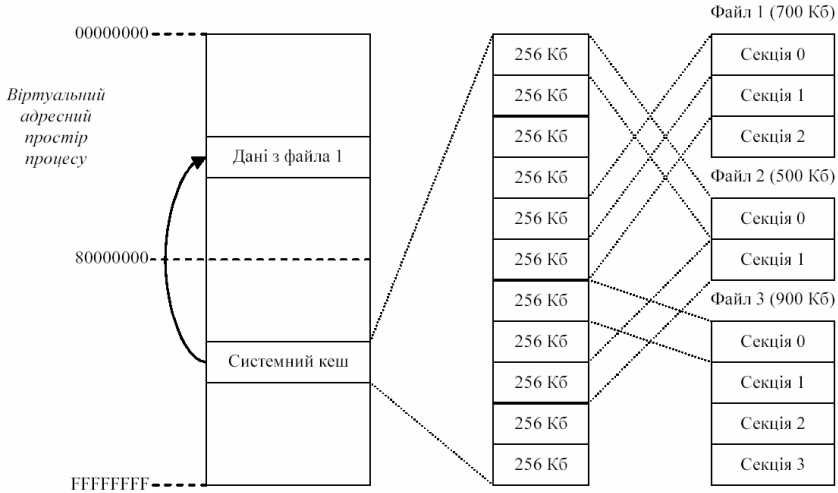


Рис. 10.9. Файли різного розміру, спроектовані в системний кеш

У результаті запит на читання з поточної позиції може бути безпосередньо задоволений із кешу. Якщо ж потрібних байтів файла в кеші немає, то файлова система обчислює логічний номер блока у файлі (LCN), потім – логічний номер блока на диску (VCN). Після цього робиться запит до системи введення-виведення на читання цього блоку, точніше, проектування в буфер кешу частини файла, що містить даний блок. Подібна організація дозволяє системі підтримувати єдиний централізований кеш для всіх використовуваних файлових систем (NTFS, FAT, CDFS, видалена FS та ін.), а файлові системи не зобов'язані управляти своїми кешами.

Важливою властивістю кешу є його когерентність. Менеджер кеша стежить за відповідністю відкритих файлів і файлів, що відображаються в пам'ять (за допомогою функції MapViewOfFile). Кожного разу, коли процес прочитує файл або відображає його в пам'ять, цей запит задовольняється з кешу шляхом копіювання відповідного блока в адресний простір процесу. Тому, незалежно від того, скільки процесів відкриють файл або відобразять його в пам'ять, реальне відображення в пам'ять відбувається один раз. У результаті всі процеси "бачитимуть" одну і ту ж версію файла.

Акуратна реалізація кешування вимагає розв'язання кількох проблем.

По-перше, місткість буфера кешу обмежена. У системного кешу немає власного робочого набору – він входить в єдиний системний робочий набір. Розміром системного робочого набору управляє менеджер пам'яті, що відповідає за його розширення або скорочення. При цьому величина, помічена як розмір системного кешу на панелі диспетчера завдань, стосується розміру всього системного робочого набору.

По-друге, оскільки кешування використовує механізм відкладеного запису (lazy write), при якому модифікація буфера не викликає негайного запису на диск, серйозною проблемою є "старіння" інформації в дискових блоках кешованого файла. Невчасна синхронізація буфера кешу і диска може призвести до дуже небажаних наслідків у випадку відмов устаткування або програмного забезпечення.

Записувані дані протягом деякого часу накопичуються, після чого скидаються на диск пакетом. З погляду швидкодії і зниження ризику від можливих втрат дуже важливий вибір частоти скидання кешу. Зазвичай підсистема відкладеного запису раз у секунду скидає на диск одну восьму кількості модифікованих сторінок системного кешу.

ОС Windows дозволяє організувати варіант синхронного режиму роботи з окремими файлами, що задається при відкритті файла, при якому всі зміни у файлі негайно зберігаються на диску. Для цього можна, наприклад, встановити прапорець FILE_FLAG_WRITE_THROUGH при виклику функції CreateFile. Фактично, це відмова від кешування і, відповідно, різке зниження продуктивності.

Крім того, у будь-який момент можна примусово скинути вміст кешу відкритого файла на диск за допомогою функції FlushFileBuffers. Нагадаємо, що для файла, що відображається, є аналогічна функція FlushViewOfFile.

Дослідження роботи програми, що ілюструє функціонування кешу

Наведена у прикладі 10.4 програма створює достатньо великий буфер пам'яті (чотири мегабайти). Потім вміст буфера записується два рази у файл. При другому записі програма скидає вміст кешу за допомогою функції FlushFileBuffers.

Приклад 10.4

```
uses windows,strings;
const N=$400000;
var i,dwCount:dword;
   pBuffer:^byte;
   hFile:HANDLE;
   d:array[0..15]of byte;
begin
for i:=0 to 15 do if i<=9 then d[i]:=i+$30 else d[i]:=i+$37;
pBuffer:=VirtualAlloc(NIL,N,MEM_RESERVE or MEM_COMMIT,
                                                PAGE_READWRITE);
for i:=0 to N-1 do (pBuffer+i)^:=d[i mod 16];
hFile:=CreateFile('Test.txt',GENERIC_READ or
GENERIC_WRITE,
FILE_SHARE_READ or FILE_SHARE_WRITE,NIL,
                                              CREATE_ALWAYS,0,0);
WriteFile(hFile,pBuffer^,N,dwCount,NIL);
CloseHandle(hFile);
hFile:=CreateFile('Test.txt',GENERIC_READ or
GENERIC_WRITE,
FILE_SHARE_READ or FILE_SHARE_WRITE,NIL,
                                                OPEN_EXISTING,0,0);
for i:=0 to N-1 do begin
WriteFile(hFile,(pBuffer+(N-1-i))^,1,dwCount,NIL);
if(i mod $40000)=0 then FlushFileBuffers(hFile);
                    end;
FlushFileBuffers(hFile);
VirtualFree(pBuffer,0,MEM_RELEASE);
CloseHandle(hFile);
end.
```

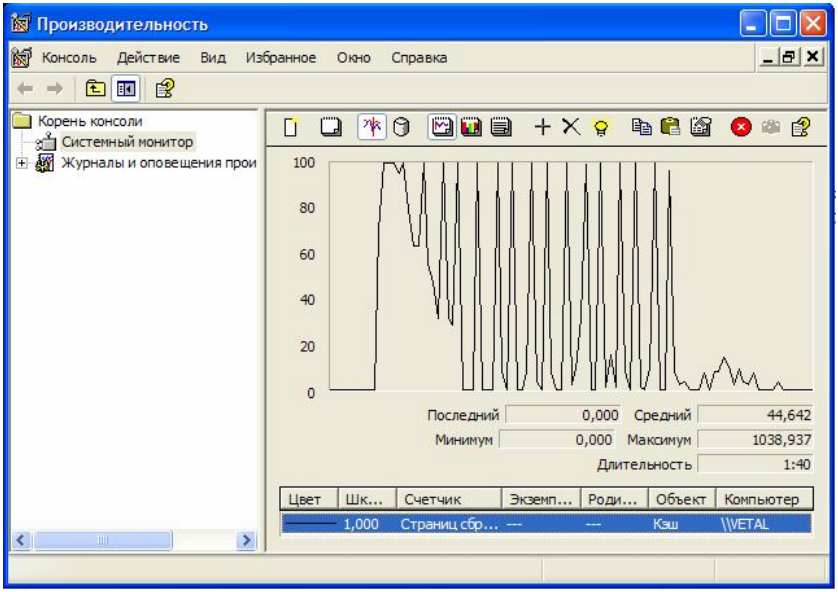


Рис. 10.10,а. Поведінка лічильника "Сторінок скидань даних / с" кешу

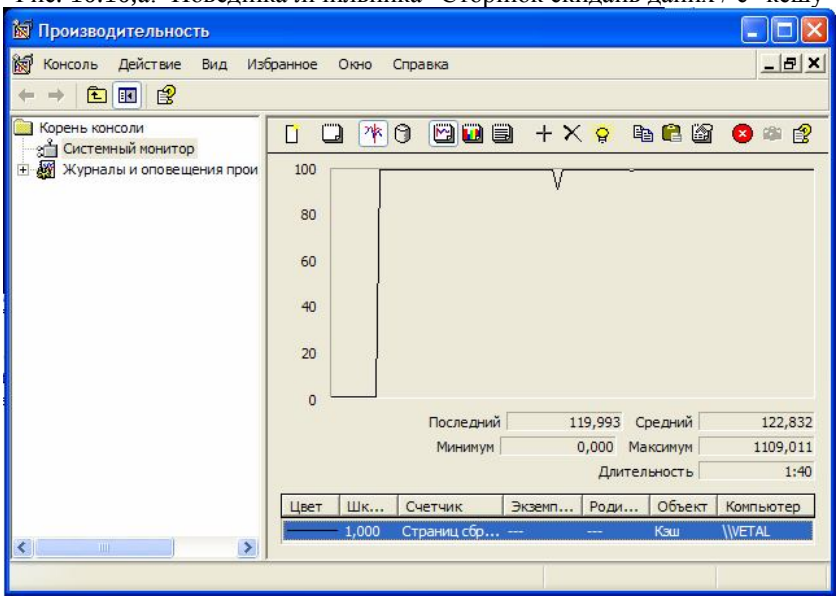


Рис. 10.10,б. Поведінка лічильника "Сторінок скидань даних / с" кешу при встановленому прапорці FILE_FLAG_WRITE_THROUGH

За результатами роботи програми можна спостерігати за допомогою лічильника "Сторінок скидань даних / с" кешу. Лічильник поводиться відповідно до рис. 10.10,*а*, де максимальні піки з'являються при першому записі буфера пам'яті у файл та при другому, услід за виконанням функції FlushFileBuffers.

На рис. 10.10,*б* показано поведінку лічильника у випадку, коли встановлено прапорець FILE_FLAG_WRITE_THROUGH при виклику функції CreateFile. Видно, що має місце постійний запис даних у файл маленькими порціями і, відповідно, різке зниження продуктивності, тобто зростає час запису.

При першому записі має місце пакетний запис буфера пам'яті, який заповнений послідовністю шістнадцятирічних цифр: 0123456789ABCDEF. При другому записі відбувається поелементний запис у файл вмісту буфера у зворотному порядку: FEDCBA9876543210. Під час запису ви можете відкривати файл Test.txt текстовим редактором і відстежувати момент останнього фактичного запису у файл, який у випадку, зображеному на рис. 10.10,*а*, збігається з викликом функції FlushFileBuffers, а у випадку, зображеному на рис. 10.10,*б* – із викликом функції WriteFile.

Рекомендується модифікувати дану програму і проаналізувати поведінку системи відкладеного скидання кешу за допомогою відповідних лічильників продуктивності.

Оптимальне розміщення інформації на диску

Кешування – *не єдиний* спосіб збільшення продуктивності системи. Інша важлива техніка – скорочення кількості рухів зчитуючої головки диска за рахунок розумної стратегії розміщення інформації. Для цього доцільно періодично здійснювати дефрагментацію диска (вичищення „сміття"). Дефрагментацію можна виконати за допомогою відповідної вкладки на адміністративній консолі панелі управління.

Надійність файлової системи

Оскільки руйнування файлової системи часто небезпечніше, ніж руйнування комп'ютера, файлові системи повинні розроблятися з урахуванням подібної можливості. Збереження інформації може бути забезпечене за рахунок її надмірності (резервне копіювання, віддзеркалювання, утворення RAID масивів). Файлові сис-

теми сучасних ОС містять спеціальні засоби для підтримки власної цілісності і несуперечності.

Відновлювана файлова система NTFS

Як правило, файлова операція зачіпає відразу кілька об'єктів файлової системи. Наприклад, запис у файл припускає виділення йому блоків диска, модифікацію MFT-записів про зайнятий простір, файл і каталог, що містить файл і т.д. Протягом короткого періоду часу між цими кроками інформація у файловій системі виявляється неузгодженою. І, якщо внаслідок непередбачуваної зупинки системи на диску будуть збережені зміни тільки для частини цих об'єктів (порушена атомарність файлової операції), файлова система на диску може бути залишена в суперечливому стані. У сучасних ОС передбачені заходи, які дозволяють звести до мінімуму збиток від псування файлової системи і потім повністю або частково відновити її цілісність.

Для позначення сукупності дій, що виконуються файловою операцією, використовується термін *транзакція*. Очевидно, що для збереження цілісності файлової системи транзакція повинна виконуватися цілком або не виконуватися взагалі.

Одним із засобів підтримки цілісності є *журналізація*. Послідовність дій з об'єктами під час транзакції протоколюється, і, якщо відбувся збій системи, то, маючи протокол, можна здійснити відкат системи назад у початковий цілісний стан, в якому вона перебувала до початку транзакції.

У журнал заносяться не всі зміни, а лише зміни метаданих (MFT-записів та ін.). Зміни в даних користувача в протокол не заносяться. Якщо протоколювати зміни даних користувача, то цим буде завдано серйозного збитку продуктивності системи, оскільки кешування втратить сенс. Витрати на журналізацію можуть бути частково компенсовані за рахунок збільшення інтервалу між скиданнями кешу на диск.

Для файла журналу відводиться другий запис таблиці MFT тому NTFS. Для мінімізації можливих згубних наслідків сервіс файла журналу (log file service, LFS) здійснює записи в такій послідовності:

1. Спочатку в кешований файл журналу заноситься запис про передбачувану транзакцію.
2. Потім робиться власне транзакція, тобто модифікується файлова система (також у кеші).
3. Після цього запис у файлі журналу скидається на диск.
4. Нарешті, скинувши на диск файл журналу, диспетчер кешу записує на диск зміни у файловій системі (результати операції над метаданими).

Описана схема – це результат компромісу між повною відмовостійкою системою і максимальною продуктивністю файлових операцій. Зрозуміло, у розпорядженні користувача залишаються засоби організації наскрізного запису на диск і примусового скидання на диск кешу NTFS (FlushFileBuffer).

Перевірка цілісності файлової системи за допомогою утиліт

Якщо порушення цілісності файлової системи все ж таки відбулося, то можна вдатися до допомоги спеціалізованих *утиліт* (chkdsk, scandisk та ін.). Вони можуть запускатися після завантаження або після збою і здійснюють багатократне сканування різноманітних структур даних файлової системи у пошуках суперечностей.

Рішення проблеми поганих блоків

Наявність дефектних блоків на диску – звичайна справа. Під "поганими" блоками зазвичай розуміють блоки диска, для яких обчислена контрольна сума зчитуваних даних не збігається з контрольною сумою, що зберігається. У NTFS застосовується один із способів нейтралізації даної проблеми – конструювання файла, що містить дефектні блоки. Для цього файла зарезервований запис 8 у таблиці MFT. У результаті „погані” блоки вилучаються зі списку вільних блоків, а отже, стають недоступними для додатків.

Фіксація змін у файловій системі

Для розв’язання ряду проблем необхідно вміти фіксувати зміни файлової системи після закінчення деякого проміжку часу. Наприклад, виявлення порушення цілісності файлів, що містять компоненти самої операційної системи, є способом захисту від

зовнішнього нав'язування і одним із засобів підвищення загальної безпеки системи. Іншим прикладом може бути актуалізація відомостей про файли в діалогах і вікнах програм, що працюють із файлами.

Для виконання подібних завдань ОС Windows має у своєму розпорядженні найширші можливості. Для знайомства з даною технікою рекомендується вивчити особливості застосування функції FindFirstChangeNotification.

Дослідження роботи програми спостереження за змінами в каталозі

У прикладі 10.5 наведено текст програми, яка відстежує зміни в деякому каталозі, в даному випадку D:\\demo_dir. Перед запуском програми створіть даний каталог і деякий текстовий файл у ньому. Після запуску програми в консольному вікні з'явиться повідомлення Wait for changes in the directory – програма очікує на зміни у директорії demo_dir. Змініть назву файла, який ви створили у директорії. При цьому в консольному вікні з'явиться повідомлення First notification: the directory was changed – фіксація першої зміни в директорії (прапорець FILE_NOTIFY_CHANGE_FILE_NAME встановлено, отже, можна відстежувати зміни назв файлів). Змініть розмір файла, відкривши його, наприклад, текстовим редактором, записавши кілька символів та зберігши на диску. При цьому в консольному вікні з'явиться повідомлення Next notification: the directory was changed – фіксація наступної зміни в директорії (прапорець FILE_NOTIFY_CHANGE_SIZE встановлено, а отже, можна відстежувати зміни розмірів файлів).

Приклад 10.5

```
uses windows;
var hChangeHandle:HANDLE;
begin
hchangeHandle:=FindFirstChangeNotification('D:\\demo_dir',TRUE,
                        FILE_NOTIFY_CHANGE_FILE_NAME or
                                FILE_NOTIFY_CHANGE_SIZE);
if(hchangeHandle=INVALID_HANDLE_VALUE)then
```

```
Writeln('Find first change notification failed.');
writeln('Wait for changes in the directory.');
if(WaitForSingleObject
                (hchangeHandle,INFINITE)=WAIT_OBJECT_0) then
writeln('First notification: the directory was changed.')
else writeln('Wait for single object failed.');
if(not FindNextChangeNotification(hChangeHandle))then
Writeln('Find next change notification failed.');
if(WaitForSingleObject
                (hChangeHandle,INFINITE)=WAIT_OBJECT_0)then
writeln('Next notification: the directory was changed.')
else writeln('Wait for single object failed.');
FindCloseChangeNotification(hChangeHandle);
end.
```

Підтримка кількох файлових систем

Подібно до багатьох сучасних операційних систем, ОС Windows підтримує кілька файлових систем (CDFS, UDF, FAT, NTFS, видалені FS). Ця можливість закладена в архітектурі системи введення-виведення. Список зареєстрованих файлових систем можна "побачити" за допомогою утиліти WinObj.

Код, що реалізовує функціональність кожної файлової підсистеми ОС, входить до складу відповідного драйвера файлової системи. Запит прикладної програми на читання (запис) із файла передається диспетчеру введення-виведення, який намагається обслужити цей запит за допомогою єдиного інтегрованого кешу (див. рис. 10.8).

Якщо потрібна інформація в кеші відсутня, диспетчер введення-виведення звертається до пристрою, що містить даний файл, підбираючи драйвер, відповідний даному пристрою. Драйвер конкретної файлової системи за ім'ям файла і байтового зсуву визначає номер блока на диску і переадресує запит на обробку цього блока драйверу диска.

Окрім кешу, драйвери файлових систем тісно інтегровані з диспетчером пам'яті, що дозволяє забезпечити коректність і когерентність механізму проектування файлів у пам'ять.

Висновок

У даній лабораторній роботі описано та досліджено окремі аспекти реалізації файлової системи NTFS. Головна функція файлової системи – зв'язок символьного імені з блоками диска – реалізована за рахунок підтримки списку блоків у записі про файл у головній файловій таблиці MFT. Для швидкого пошуку файла по імені каталог може бути організований у вигляді B+ дерева. Проблеми монтування дисків і скріплення файлів розв'язуються за допомогою точок повторного аналізу. Продуктивність файлової системи забезпечується менеджером кеша, а також шляхом оптимального розміщення інформації на диску. Для відновлення системи після відмови живлення ведеться журнал файлових операцій із метаданими. Підтримка декількох файлових систем в ОС Windows забезпечується оригінальною структурою підсистеми введення-виведення, в рамках якої для кожної файлової системи є відповідний драйвер.

Практична частина

1. Складіть програми, наведені у прикладах 10.1 – 10.5 теоретичної частини даної лабораторної роботи. Запустіть програми та переконайтеся, що вони працюють правильно (згідно з описом алгоритму їх роботи).

2. На основі наведених програм дослідіть різноманітні особливості роботи сервісних API-функцій файлової системи NTFS операційної системи Windows.

3. Складіть програми, рекомендовані як самостійні вправи.

4. Оформіть звіт із виконаної лабораторної роботи, який повинен містити текст програм, демонстрацію результатів роботи, та дайте відповіді на контрольні запитання.

Контрольні запитання та завдання

1. Що таке кластер?
2. Як розміри кластерів впливають на функціонування файлової системи?
3. В який спосіб можна отримати інформацію про кластери?
4. Що собою являє головна файлова таблиця MFT (master file table)?
5. Як виконується завдання приведення у відповідність номера блока у файлі (LCN – logical cluster number) номеру блоку на диску (VCN – volume cluster number)?
6. Що собою являє запис у таблиці MFT для каталогу (директорії)?
7. Як відбувається процес пошуку файла по імені?
8. Які API-функції пошуку файлів ви знаєте? Опишіть особливості їх використання.
9. Опишіть процес монтування файлових систем.
10. Обґрунтуйте необхідність використання механізму захоплення частини файла для монопольного доступу.
11. Що таке кешування?
12. Опишіть особливості функціонування менеджера кешу ОС Windows.
13. Як забезпечується надійність та відновлюваність файлової системи NTFS?
14. В який спосіб відбувається фіксація змін у файловій системі NTFS?
15. Які API-функції для спостереження за змінами у каталозі ви знаєте? Опишіть особливості їх використання.

Лабораторна робота № 11
Дослідження системи керування дискреційним доступом у операційній системі Windows

Теоретична частина

Вступ

Відомо, що одним із найважливіших компонентів системи безпеки ОС Windows є система контролю й управління дискреційним доступом. Для її опису прийнято використовувати формальні моделі. Хоча застосування формальних моделей захищеності не дозволяє строго обґрунтувати безпеку інформаційних систем (ІС) для ряду найцікавіших випадків, вони формують корисний понятійний апарат, який може бути застосований для декомпозиції і аналізу досліджуваної системи.

Для побудови формальних моделей безпеки прийнято зображувати ІС у вигляді сукупності взаємодіючих сутностей – *суб'єктів* (s) і *об'єктів* (o).

Об'єкти Windows, що захищаються, включають: файли, пристрої, канали, події, мютекси, семафори, розділи спільної пам'яті, розділи реєстру і ряд інших. Сутність, від якої потрібно захищати об'єкти, називається "суб'єктом". Суб'єктами в Windows є процеси і потоки, що запускаються конкретними користувачами. Суб'єкт безпеки – активна системна складова, а об'єкт – пасивна.

Крім дискреційного доступу, Windows підтримує управління *привілейованим* доступом. Це означає, що в системі є користувач-адміністратор із необмеженими правами. Для спрощення адміністрування (а також для відповідності стандарту POSIX) користувачі Windows об'єднані в групи. Приналежність до групи пов'язана з певними привілеями, наприклад із привілеєм вимикати комп'ютер. Користувач, як член групи, володіє, таким чином, набором повноважень, необхідних для його діяльності, і відіграє певну роль. Подібна стратегія називається управлінням ролевим доступом.

Для того, щоб з'ясувати, якою мірою комбінація у вигляді управління дискреційним і ролевим доступом служить гарантією захисту для виконуваних програм, бажано мати уявлення про

формальні моделі, використовувані при побудові системи безпеки ОС Windows. Можливості формальних моделей проаналізовані в додатку до даної лабораторної роботи.

Основний висновок з аналізу використовуваних в ОС Windows моделей контролю доступу (комбінація дискреційної і ролевої): не можна формально обґрунтувати безпеку ІС у випадках, що являють практичний інтерес. Необхідно обґрунтовувати безпеку конкретної системи шляхом її активного дослідження.

Ключова мета системи захисту Windows – стежити за тим, хто і до яких об'єктів здійснює доступ. Система захисту зберігає інформацію, що належить до безпеки для кожного користувача, групи користувачів і об'єкта. Модель захисту ОС Windows вимагає, щоб суб'єкт на етапі відкриття об'єкта вказував, які операції він збирається виконувати відносно цього об'єкта. Одноманітність контролю доступу до різних об'єктів (процесів, файлів, семафорів та ін.) забезпечується тим, що з кожним процесом (потоком) пов'язаний маркер доступу, а з кожним об'єктом – дескриптор захисту. Маркер доступу як параметр має ідентифікатор користувача, а дескриптор захисту – списки прав доступу. ОС може контролювати спроби доступу, які прямо або опосередковано здійснюються процесами і потоками, ініційованими користувачем.

ОС Windows відстежує і контролює доступ до різноманітних об'єктів системи (файли, принтери, процеси, іменовані канали і т.д.). Окрім дозвільних записів, списки прав доступу містять і забороняючи записи, щоб користувач, якому доступ до об'єкта заборонений, не зміг одержати його як член якої-небудь групи, якій цей доступ наданий.

Користувачі системи для спрощення адміністрування (а також для відповідності стандарту POSIX) об'єднані в групи. Користувачів і групи іноді називають учасниками безпеки. Користувачі за допомогою породжуваних ними суб'єктів (процесів, потоків) здійснюють доступ до об'єктів (файлів, пристроїв тощо). Вивчення моделі контролю доступу ОС Windows доцільно почати з аналізу характеристик суб'єктів і об'єктів, істотних для організації дискреційного доступу.

Інструментальні засоби управління безпекою

Перш ніж почати вивчення API системи, доцільно ознайомитися з корисними утилітами й інструментальними засобами.

Для управління системою безпеки в ОС Windows є різноманітні і зручні інструментальні засоби. Зокрема, в рамках даної теми буде потрібно вміння управляти обліковими записами користувачів за допомогою панелі "Користувачі і паролі". Крім того, знадобиться контролювати привілеї користувачів за допомогою панелі "Призначення прав користувачам". Рекомендується також освоїти роботу з утилітою переглядання даних маркера доступу процесу WhoAmI.exe, утилітами перегляду і редагування списків контролю доступу (cacls.exe, ShowACLs.exe, SubInACL,exe, SvcACL.exe), утилітою переглядання маркера доступу процесу PuList.exe і рядом інших.

Велика кількість інтерактивних засобів не усуває необхідності програмного управління різними об'єктами в середовищі ОС Windows. Застосування API системи дозволяє краще вивчити її особливості і створювати додатки, відповідні складним вимогам захисту. Прикладом можуть служити різні сценарії обмеження доступу (застосування обмежених маркерів доступу, перевтілення, створення об'єктів, не пов'язаних із конкретним користувачем, і т.д.). Проте вбудовані інструментальні можливості системи активно використовуватимуться як допоміжні засоби при розробці різноманітних програмних додатків.

Користувачі і групи користувачів

Кожен користувач (і кожна група користувачів) системи повинен мати обліковий запис (account) у базі даних системи безпеки. Облікові записи ідентифікуються ім'ям користувача і зберігаються в базі даних SAM (Security Account Manager) у розділі HKLM/SAM реєстру.

Обліковий запис користувача містить набір відомостей про користувача, такі як ім'я, пароль (або реквізити), коментарі і адреса. Найбільш важливими елементами облікового запису користувача є: список привілеїв користувача відносно даної системи, список груп, до яких належить користувач, та ідентифікатор безпеки **SID** (Security IDentifier). Ідентифікатори безпеки генерують-

ся при створенні облікового запису. Вони (а не імена користувачів, які можуть не бути унікальними) служать основою для ідентифікації суб'єктів внутрішніми процесами ОС Windows.

Облікові записи груп, створені для спрощення адміністрування, містять список облікових записів користувачів, а також включають відомості, аналогічні відомостям облікового запису користувача (SID групи, привілеї члена групи та ін.).

Створення облікового запису користувача

Основним засобом створення облікового запису користувача служить Win32-функція NetUserAdd, що належить сімейству мережних (*Net*) функцій ОС Windows, докладний опис якої є в MSDN. За допомогою Net-функцій можна управляти обліковими записами користувачів як на локальній, так і на віддаленій системі (детальніше про використання Net-функцій можна прочитати у відповідній літературі).

Для успішного застосування *Net-функцій* досить знати таке. По-перше, Net-функції входять до складу бібліотеки NetApi32, яку потрібно у явний спосіб додати до проекту, а прототипи функцій оголошені в заголовному файлі Lm. По-друге, Net-функції підтримують рядки тільки у форматі Unicode. Нарешті, інформацію про обліковий запис Net-функції потрібно передавати за допомогою спеціалізованих структур, найменш складна з яких структура USER_INFO_1.

Дослідження роботи програми створення нового облікового запису

Для ілюстрації розглянемо нескладну програму, завдання якої – створити новий обліковий запис для користувача "ExpUser".

Приклад 11.1

```
uses windows,lm;
function
CreateUser(pszName:PWSTR;pszPassword:PWSTR):boolean;
var ui:USER_INFO_1;
   nStatus:NET_API_STATUS;
begin
ui.usri1_password_age:=0;
```

```
ui.usri1_home_dir:=nil;
ui.usri1_comment:=nil;
ui.usri1_flags:=0;
ui.usri1_script_path:=nil;
  ui.usri1_name := pszName;                    // ім'я користувача
  ui.usri1_password := pszPassword;            // пароль користувача
  ui.usri1_priv := USER_PRIV_USER;             // звичайний користувач
  nStatus := NetUserAdd(NIL,1,PBYTE(@ui),NIL);
  if nStatus=NERR_Success then CreateUser:=true else
CreateUser:=false;
end;
begin
   if not CreateUser('ExpUser','123')then
      writeln('A system error has occurred.');
end.
```

Результат роботи програми – створення нового користувача – можна проконтролювати за допомогою аплета панелі управління "Локальні користувачі". Після створення користувача доцільно наділити його мінімальним набором прав, наприклад правом входу у систему. Найрозумніше – включити користувача в якунебудь групу, наприклад у групу звичайних користувачів. У цьому випадку знов створений користувач одержить привілеї члена даної групи. Це можна зробити за допомогою того ж аплета. Про те, як забезпечити користувача необхідними привілеями у програмний спосіб, буде розказано нижче.

Для видалення облікового запису користувача використовується функція NetUserDel.

На основі попередньої програми, як самостійний приклад, рекомендується написати програму видалення облікового запису конкретного користувача.

На закінчення даного розділу хотілося б ще раз підкреслити, що хоча *Net-функції* дозволяють працювати з іменами облікових записів, решта частин системи для ідентифікації облікового запису використовує ідентифікатор безпеки SID.

Ідентифікатор безпеки SID

Структура ідентифікатора безпеки

SID користувача (і групи) є унікальним внутрішнім ідентифікатором і структурою змінної довжини з коротким заголовком, за яким іде довге випадкове число. Це числове значення формується з ряду параметрів, причому стверджується, що ймовірність появи двох однакових SID практично дорівнює нулю. Зокрема, якщо видалити користувача в системі, а потім створити його під тим же ім'ям, то SID знов створеного користувача буде вже іншим.

Дізнатися свій ідентифікатор безпеки користувач легко може за допомогою утиліт *whoami* або *getsid* з ресурсів Windows. Наприклад, так:

```
> whoami /user /sid
```

За допомогою команди whoami /all можна одержати всю інформацію з маркера доступу процесу (див. наступні розділи).

Як самостійне завдання виконайте таку послідовність дій:

1. Створіть обліковий запис користувача за допомогою інструментальних засобів ОС Windows.
2. З'ясуйте значення його SID'а.
3. Потім видаліть цей обліковий запис і знов створіть під тим же ім'ям.
4. Порівняйте SID нового користувача з попереднім значенням SID'а.

Система зберігає ідентифікатори безпеки в бінарній формі, проте існує і текстова форма зображення SID. Текстова форма використовується для виведення поточного значення SID, а також для інтерактивного введення (наприклад, у реєстр).

У текстовій формі кожен ідентифікатор безпеки має певний формат. Спочатку знаходиться префікс S, за яким розміщується група чисел, розділених дефісами. Наприклад, SID адміністратора системи має вигляд: S-1-5-<домен>-500, а SID групи everyone, в яку входять всі користувачі, включаючи анонімних і гостей, – S-1-1-0.

Оскільки структура SID має змінну довжину, для копіювання SID і перетворення його в текстову форму і назад рекомендується

вдаватися до спеціальних функцій CopySID, ConvertSidToStringSid і ConvertStringSidToSid.

Щоб набути бінарного значення SID на ім'я користувача, потрібно використовувати функцію LookupAccountName. Зворотне завдання може бути виконане за допомогою функції LookupAccountSid.

Дослідження роботи програми отримання ідентифікатора безпеки

Як ілюстрацію складіть просту програму, завдання якої – отримання значення Sid для поточного облікового запису і перетворення його в текстову форму.

У даній програмі спочатку повинні формуватися параметри виклику функції LookupAccountName. Зокрема, ім'я поточного облікового запису повертає функція GetUserName. Крім того, необхідно виділити пам'ять для імені домена. Результуюче значення SID можна порівняти з тим, яке видає утиліта Whoami.exe.

На основі попередньої програми, як самостійну вправу, рекомендується написати програму, яка за допомогою функції LookupAccountSid вирішує зворотну задачу – дозволяє з'ясувати ім'я власника даного ідентифікатора безпеки.

Об'єкти. Дескриптор захисту

У ОС Windows всі типи об'єктів захищені однаковим чином. Із кожним об'єктом пов'язаний *дескриптор захисту (security descriptor)*. Дескриптор захисту описується структурою типу SECURITY_DESCRIPTOR та ініціюється функцією InitializeSecurityDescriptor.

Зв'язок об'єкта з дескриптором відбувається в момент створення об'єкта. Наприклад, один з аргументів функції CreateFile – покажчик на структуру SECURITY_ATTRIBUTES, яка містить покажчик на дескриптор захисту.

Дескриптор захисту (див. рис. 11.1) містить SID власника об'єкта, SID груп для даного об'єкта і два покажчики на списки DACL (Discretionary ACL) і SACL (System ACL) контролю доступу. DACL і SACL містять дозвільні й заборонні щодо доступу списки користувачів і груп, а також списки користувачів, чиї спроби доступу до даного об'єкта підлягають аудиту.

Структура кожного ACL списку проста. Це набір записів ACE (Access Control Entry), кожен запис містить SID і перелік прав, наданих суб'єкту із цим SID.

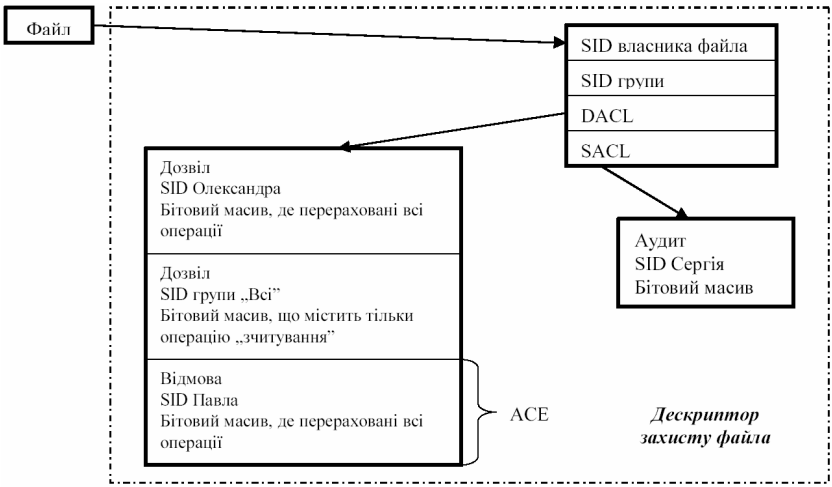


Рис. 11.1. Структура дескриптора захисту для файла

У списку ACL є записи ACE двох типів – ті, які дозволяють і забороняють доступ. Дозвільний запис містить SID користувача або групи і бітовий масив (access mask), що визначає набір операцій, які процеси, що запускаються цим користувачем, можуть виконувати з даним об'єктом. Заборонний запис діє аналогічно, але в цьому випадку процес не може виконувати перераховані операції. Бітовий масив, або маска доступу, складається з 32 бітів і зазвичай формується програмним способом із певним чином визначених констант, описаних у файлах-заголовках компілятора (переважно у файлі WinNT.h). Формат маски доступу можна подивитися у відповідній літературі.

На прикладі, зображеному на рис. 11.1, власник файла Олександр має право на всі операції з даним файлом, іншим зазвичай дається тільки право на читання, а Павлу заборонені всі операції. Таким чином, список DACL описує всі права доступу до об'єкта. Якщо цього списку немає, то всі користувачі мають всі права;

якщо цей список існує, але він порожній, права має тільки його власник.

Окрім списку DACL, дескриптор захисту включає також список SASL, який має таку ж структуру, що й DACL, тобто складається з таких самих ACE-записів, тільки замість операцій, що регламентують доступ до об’єкта, в ньому перераховані операції, що підлягають аудиту. У прикладі на рис. 11.1 операції з файлом процесів, що запускаються Сергієм, описані у відповідному бітовому масиві, реєструватимуться в системному журналі.

Як самостійне завдання сформуйте список прав доступу для файла за допомогою інструментальних засобів ОС Windows.

Для установки прав доступу до файла, що знаходиться на NTFS розділі диска, потрібно вибрати вкладку "Безпека" аплета "Властивості", який виникає в Windows Explorer при натисненні на ярлик файла правою кнопкою миші.

Отже, дескриптор захисту має достатньо складну структуру і його формування виглядає непростим завданням. На щастя, в Windows є стандартний механізм, що призначає доступ до об’єктів "за замовчуванням", якщо додаток не поклопотався створити його явно. В таких випадках кажуть, що об’єкту призначений стандартний захист. Прикладом може служити створення файла за допомогою функції CreateFile, де параметрові-покажчику на структуру SECURITY_ATTRIBUTES присвоєне значення NIL. Деякі об’єкти використовують тільки стандартний захист (м’ютекси, події, семафори).

Суб’єкти зберігають інформацію про стандартний захист, який буде призначено створюваним об’єктам, у своєму маркері доступу (див. наступний розділ). Зрозуміло, в ОС Windows є всі необхідні засоби для настройки стандартного захисту, зокрема списку DACL "за замовчуванням", в маркері доступу суб’єкта.

Основне джерело інформації про захист об’єкта – Win32-функція GetSecurityInfo, тоді як настройка захисту об’єкта може бути здійснена за допомогою функції SetSecurityInfo.

Дослідження роботи програми отримання інформації з дескриптора захисту файла

Як приклад складіть програму, завдання якої – отримання текстового значення Sid власника файла з дескриптора захисту файла.

Робота даної програми повинна полягати у відкритті існуючого файла MyFile.txt і застосуванні функції GetSecurityInfo для витягування ідентифікатора безпеки власника з дескриптора захисту файла. Потім ідентифікатор перетворюється в текстову форму і виводиться на екран.

На основі попередньої програми, як самостійну вправу, рекомендується написати програму, яка виводить на екран список прав доступу до позначеного файла.

Суб'єкти безпеки. Процеси, потоки. Маркер доступу

Так само як і об'єкти, суб'єкти повинні мати відмітні ознаки – контекст користувача, для того, щоб система могла контролювати їх дії. Відомості про контекст користувача зберігаються в маркері (вживаються також терміни "токен", "жетон") доступу. При інтерактивному вході в систему користувач зазвичай вводить своє ім'я і пароль. Система (процедура Winlogon) по імені знаходить відповідний обліковий запис, витягує з неї необхідну інформацію про користувача, формує список привілеїв, що асоціюються з користувачем і його групами, і все це об'єднує в структуру даних, яка називається маркером доступу. Маркер також зберігає деякі параметри сесії, наприклад час закінчення дії маркера. Отже, саме маркер є тією візитною карткою, яку суб'єкт повинен пред'явити, щоб здійснити доступ до якого-небудь об'єкта.

Услід за оболонкою (Windows Explorer) усі процеси (а також всі потоки процесу), що запускаються користувачем, успадковують цей маркер. Коли один процес створює інший за допомогою функції CreateProcess, дочірньому процесу передається дублікат маркера, який, таким чином, розповсюджується по системі.

Основні компоненти маркера доступу показані на рис. 11.2.

SID користувача	SID_1, … SID_n Ідентифікатори груп користувача	DACL за замовчуванням	Привілеї	Інші параметри

Рис. 11.2. Основні компоненти маркера доступу

Включаючи в маркер інформацію про захист, зокрема DACL, Windows спрощує створення об’єктів зі стандартними атрибутами захисту. Як уже мовилося, якщо процес не потурбується про те, щоб явним чином вказати атрибути безпеки об’єкта, на підставі списку DACL, наявного в маркері, будуть сформовані права доступу до об’єкта за замовчуванням. Налаштовування стандартного захисту можна здійснити за допомогою функції SetTokenInformation. При цьому, оскільки об’єкти в Windows відрізняються великою різноманітністю, в списку DACL "за замовчуванням" можна вказати тільки так звані базові права доступу, з яких система формуватиме стандартні права доступу залежно від виду створюваного об’єкта. Те, як це робиться і як сформувати список DACL, за замовчуванням наявний у маркері, детально описано у відповідній літературі.

Як самостійне завдання здійсніть переглядання даних маркера доступу за допомогою утиліти WhoAmi.

Якщо процес володіє правом TOKEN_QUERY доступу до об’єкта, то велику частину вмісту маркера можна отримати за допомогою функції GetTokenInformation. Щоб одержати описувач маркера, необхідно скористатися функцією OpenProcessToken.

Як приклад складіть програму, завдання якої – отримання текстового значення SID власника процесу з маркера доступу даного процесу.

Робота даної програми повинна полягати в одержуванні описувача поточного процесу за допомогою функції OpenProcessToken і застосуванні функції GetTokenInformation для витягування з нього ідентифікатора безпеки власника. Після цього ідентифікатор перетворюється в текстову форму і виводиться на екран.

У наступній лабораторній роботі буде наведений приклад програми отримання з маркера доступу інформації про привілеї користувача.

На основі попередньої програми, як самостійне завдання, рекомендується написати програму, яка виводить на екран список прав доступу "за замовчуванням" для об’єктів, що створюються даним процесом.

Перевірка прав доступу

Після формалізації атрибутів захисту суб'єктів і об'єктів можна перерахувати основні етапи перевірки прав доступу див. рис. 11.3.

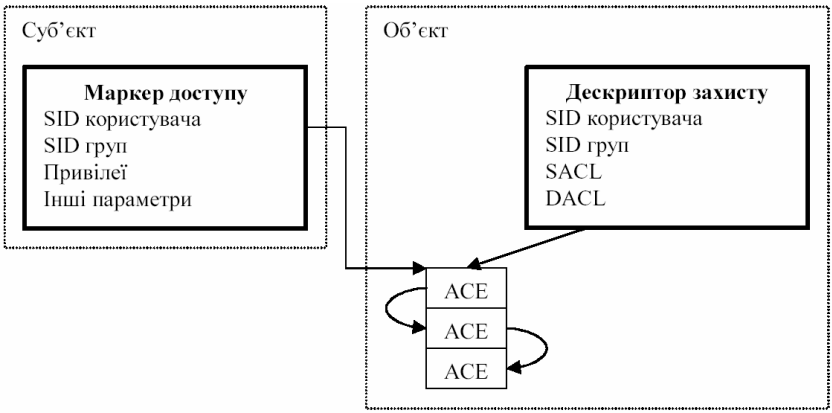


Рис. 11.3. Приклад перевірки прав доступу до захищеного об'єкта

Етапів перевірки досить багато. Найбільш важливі етапи з них такі:

- Якщо SID суб'єкта збігається із SID власника об'єкта й запрошуються стандартні права доступу, то доступ надається незалежно від вмісту DACL.
- Далі система послідовно порівнює SID кожного ACE з DACL із SID маркера. Якщо виявляється відповідність, виконується порівняння маски доступу з правами, що перевіряються. Для забороняючих ACE навіть при частковому збігу прав доступ негайно відхиляється. Для успішної перевірки дозволяючих елементів необхідний збіг усіх прав.

Очевидно, що для процедури перевірки важливий порядок розташування ACE в DACL. Тому Microsoft пропонує так званий переважний порядок розміщення ACE. Наприклад, для прискорення рекомендується розміщувати заборонні елементи перед дозвільними.

Висновок

Підсистема захисту даних є однією з найбільш важливих. У центрі системи безпеки ОС Windows знаходиться система контролю доступом. Реалізовані моделі дискреційного і ролевого доступу зручні й широко розповсюджені, проте не дозволяють формально обґрунтувати безпеку додатків у ряді випадків, що представляють практичний інтерес. Із кожним процесом або потоком, тобто активним компонентом (суб'єктом), зв'язаний маркер доступу, а в кожного об'єкта (наприклад, файла), що захищається, є дескриптор захисту. Перевірка прав доступу зазвичай здійснюється у момент відкриття об'єкта і полягає в зіставленні прав суб'єкта списку прав доступу, який зберігається у складі дескриптора захисту об'єкта.

Додаток.

Формальні моделі захищеності в ОС Windows

Подання інформаційної системи як сукупності взаємодіючих сутностей – суб'єктів і об'єктів – є базовим представленням більшості формальних моделей. Захисту потребують і об'єкти, і суб'єкти. У цьому сенсі суб'єкт є окремим випадком об'єкта. Іноді кажуть, що суб'єкт – це об'єкт, який здатний здійснювати перетворення даних і якому передано управління.

Застосування формальної моделі будується на привласненні суб'єктам і об'єктам ідентифікаторів і фіксації набору правил, що дозволяють визначити, чи має даний суб'єкт авторизацію, достатню для надання вказаного типа доступу до даного об'єкта.

Застосування даного підходу може бути проілюстровано так.

Є сукупність об'єктів $\{O_i\}$, суб'єктів $\{S_i\}$ і користувачів $\{U_i\}$. Вводиться операція доступу $\{S_i\} \rightarrow \{O_j\}$, під якою мається на увазі використання і-м суб'єктом інформації з j-го об'єкта. Основні варіанти доступу: читання, запис і активація процесу, записаного в об'єкті. В результаті останньої операції з'являється новий суб'єкт $\{S_i\} \rightarrow \{O_j\} \rightarrow \{S_k\}$.

Під час завантаження створюється ряд суб'єктів (системних процесів), до яких належить оболонка $S_{shell,}$ за допомогою якої користувачі, що пройшли аутентифікацію, можуть створювати свої суб'єкти (запускати свої програми). За допомогою виконува-

виконуваних програм користувачі здійснюють доступ до об'єктів (наприклад, здійснюють читання файлів). У результаті діяльність такого користувача може бути описана орієнтованим графом доступу (див. рис. 11.4).

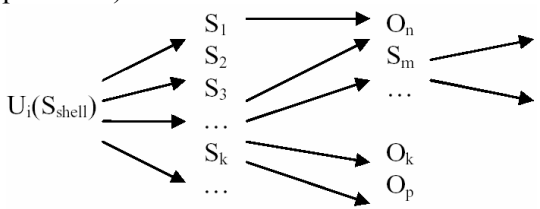


Рис. 11.4. Приклад графа доступу

Множину графів доступу можна розглядати як фазовий простір, а функціонування конкретної системи – як траєкторію у фазовому просторі. Захист інформації може полягати в тому, щоб уникати "несприятливих" траєкторій. Практично таке управління можливе тільки обмеженням на доступ у кожен момент часу, або, іншими словами, всі питання безпеки інформації визначаються описом доступів суб'єктів до об'єктів.

Можна ввести також особливий вид суб'єкта, який активізується при кожному доступі й називається монітором звернень. Якщо монітор звернень у змозі відрізнити легальний доступ від нелегального і не допустити останнього, то такий монітор називається *монітором безпеки (МБ)* і є одним із найважливіших компонентів системи захисту.

Дискреційна модель контролю і управління доступом

Моделі управління доступом регламентують доступ суб'єктів до об'єктів. Найбільш поширена так звана дискреційна (довільна) модель, в якій звичайні користувачі можуть брати участь у визначенні функцій політики і привласненні атрибутів безпеки. Серед дискреційних моделей класичною вважається модель Харрісона-Руззо-Ульмана – в ній система захисту представлена у вигляді набору множин, елементами яких є складові частини системи захисту: суб'єкти, об'єкти, рівні доступу, операції і т.п.

З концептуальної точки зору поточний стан прав доступу при дискреційному управлінні описується матрицею (див. рис. 11.5),

в рядках якої перераховані суб'єкти, в стовпцях – об'єкти, а в комірках – операції, які суб'єкт може виконати над об'єктом. Операції залежать від об'єктів. Наприклад, для файлів це операції читання, запису, виконання, зміни атрибутів, а для принтера – операції друку і управління. Поведінка системи моделюється за допомогою поняття стану Q = (S, O, M), де S, O, M – відповідно поточна множина суб'єктів, об'єктів і поточний стан матриці доступу.

Об'єкт / Суб'єкт	O_1	O_2	O_3	O_4(Printer)
S_1	Read			
S_2				Print
S_3		Read	Execute	
S_4	Read/Write		Read/Write	

Рис. 11.5. Приклад матриці доступу

З урахуванням дискреційної моделі можна в такий спосіб сформулювати завдання системи захисту інформації. З погляду безпеки, поведінка системи може моделюватися як послідовність станів, описуваних сукупністю суб'єктів, об'єктів і матрицею доступу. Тоді можна стверджувати, що для безпеки системи в стані Q_0 не повинно існувати послідовності команд, в результаті якої право R буде занесено в елемент пам'яті матриці доступу M, де воно було відсутнє в стані Q_0. (критерій Харрісона-Руззо-Ульмана). По суті, необхідно відповісти на запитання: чи зможе деякий суб'єкт S коли-небудь отримати які-небудь права доступу до деякого об'єкта O?

Очевидно, що для забезпечення безпеки необхідно накласти заборону на деякі відносини доступу. Харрісон, Руззо і Ульман довели, що в загальному випадку не існує алгоритму, який може для довільної системи, її початкового стану Q_0 і загального права r вирішити, чи ця конфігурація безпечна. Щоб задовольнити критерію безпеки в загальному випадку, в інформаційній системі повинні бути відсутніми деякі операції створення і видалення сутності, внаслідок чого експлуатація подібної системи втрачає практичний сенс.

Канали просочування інформації в системах із дискреційним доступом

Таким чином, якщо не вживати спеціальні заходи, система з дискреційним доступом виявляється незахищеною. Для ілюстрації можна продемонструвати організацію каналу витоку.

Очевидно, що для кожного суб'єкта S, активізованого у момент часу t, існує єдиний активізований суб'єкт S', який активізував суб'єкт S. Тому можна, аналізуючи граф, подібний до зображеного на рис. 11.5, виявити єдиного користувача, від імені якого активізований суб'єкт S.

Відповідно, для будь-якого об'єкта існує єдиний користувач, від імені якого активізований суб'єкт, що створив даний об'єкт. Можна сказати, що цей користувач породив даний об'єкт, а також сформував для нього список прав доступу.

В цьому випадку схемне зображення каналу витоку у вигляді дозволеного доступу від імені різних користувачів до одного і того ж об'єкта виглядатиме так:

$U_i \rightarrow$ (Write) $\rightarrow$ O, у момент часу t_1 і $U_j \rightarrow$ (Read) $\rightarrow$ O у момент часу t_2, де $i \neq j$ і $t_1 < t_2$,

Фактично це означає, що ніщо не заважає легальному користувачу (наприклад, унаслідок програмної помилки) перекинути секретну інформацію в знов створений об'єкт, доступ до якого відкритий всім охочим, або організувати прихований канал витоку за допомогою троянського коня.

Отже, формальний аналіз показує, що безпечне використання систем із дискреційним контролем доступу припускає наявність додаткових захисних заходів, а саме – ретельне проектування і коректну реалізацію системи захисту.

Ролева політика безпеки

Інша можливість, ОС Windows, що надається, – управління ролевим, зокрема привілейованим доступом. Ролева політика призначена в першу чергу для спрощення адміністрування інформаційних систем із великою кількістю користувачів і різних ресурсів. У ролевій політиці управління доступом здійснюється за допомогою правил. Спочатку для кожної ролі вказується набір привілеїв по відношенню до системи і повноважень, що являють

собою набір прав доступу до об'єктів, і вже потім кожному користувачу призначається список доступних йому ролей. Класичне поняття суб'єкт заміщається поняттями *користувач* і *роль*. Прикладом ролі є присутня майже в кожній системі роль суперкористувача (група Administrator для ОС Windows). Кількість ролей зазвичай не відповідає кількості реальних користувачів – один користувач може виконувати декілька ролей, і навпаки, декілька користувачів можуть в рамках однієї і тієї ж ролі виконувати типову роботу.

Отже, в рамках ролевої моделі формуються близькі до реального життя правила контролю доступу і обмеження, дотримання яких і служить критерієм безпеки системи. Проте, як і у випадку дискреційної моделі, ролева політика безпеки не гарантує безпеку за допомогою формальних доказів.

Практична частина

1. Складіть програму, наведену у прикладі 11.1 теоретичної частини даної лабораторної роботи. Запустіть програму та переконайтеся, що вона працює правильно (згідно з описом алгоритму її роботи).

2. Складіть програми, рекомендовані як самостійні вправи.

4. Оформіть звіт із виконаної лабораторної роботи, який повинен містити текст програм, демонстрацію результатів роботи, та дайте відповіді на контрольні запитання.

Контрольні запитання та завдання

1. У чому полягає мета системи захисту Windows?
2. Які ви знаєте інструментальні засоби управління безпекою?
3. Що таке обліковий запис користувача і з яких елементів він складається?
4. Опишіть структуру ідентифікатора безпеки.
5. Що таке дескриптор захисту?
6. Опишіть структуру дескриптора захисту для файла.
7. Що таке маркер доступу?
8. Опишіть основні компоненти маркера доступу.
9. Опишіть перевірку прав доступу до захищеного об'єкта.
10. Що таке дискреційна модель контролю і управління доступом?

Лабораторна робота № 12
Дослідження структури системи захисту операційної системи Windows

Теоретична частина

Основні компоненти системи безпеки ОС Windows

У даній лабораторній роботі будуть розглянуті питання структури системи безпеки, особливості ролевого доступу і декларована політика безпеки системи.

Система контролю дискреційного доступу – центральна концепція захисту ОС Windows, проте перелік завдань, що виконуються для забезпечення безпеки, цим не вичерпується. У даному розділі будуть проаналізовані структура, політика безпеки і API системи захисту.

Вивчення структури системи захисту допомагає зрозуміти особливості її функціонування. Незважаючи на слабку документованість ОС Windows за непрямими джерелами, можна судити про особливості її функціонування.

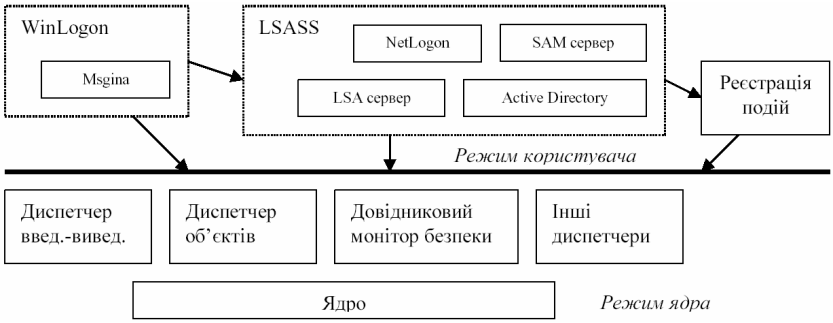


Рис. 12.1. Структура системи безпеки ОС Windows

Система захисту ОС Windows складається з таких компонентів (див. рис. 12.1).

- Процедура реєстрації (Logon Processes), яка обробляє запити користувачів на вхід у систему. Вона включає початкову інтерактивну процедуру, що відображає початковий діалог із користувачем на екрані, і віддалені процедури входу, які дозволя-

ють віддаленим користувачам дістати доступ з робочої станції мережі до серверних процесів Windows NT. Процес Winlogon реалізований у файлі Winlogon.exe і виконується як процес режиму користувача. Стандартна бібліотека аутентифікації Gina реалізована у файлі Msgina.dll.

- Підсистема локальної авторизації (Local Security Authority, LSA), яка гарантує, що користувач має дозвіл на доступ в систему. Цей компонент – центральний для системи захисту Windows NT. Він породжує маркери доступу, управляє локальною політикою безпеки і надає інтерактивним користувачам аутентифікаційні послуги. LSA також контролює політику аудиту і веде журнал, в якому зберігаються повідомлення, що породжуються диспетчером доступу. Основна частина функціональності реалізована в Lsasrv.dll.
- Менеджер обліку (Security Account Manager, SAM), який управляє базою даних обліку користувачів. Ця база даних містить інформацію про всіх користувачів і групи користувачів. Дана служба реалізована в Samsrv.dll і виконується в процесі Lsass.
- Диспетчер доступу (Security Reference Monitor, SRM), який перевіряє, чи має користувач право на доступ до об'єкта і на виконання тих дій, які він намагається зробити. Цей компонент забезпечує легалізацію доступу і політику аудиту, визначувані LSA. Він надає послуги для програм супервізорного режиму і режиму користувача, щоб гарантувати, що користувачі і процеси, що здійснюють спроби доступу до об'єкта, мають необхідні права. Даний компонент також породжує повідомлення служби аудиту, коли це необхідно. Це компонент виконавчої системи: Ntoskrnl.exe.

Всі компоненти активно використовують базу даних Lsass, що містить параметри політики безпеки локальної системи, яка зберігається в розділі HKLM\SECURITY реєстру.

Як уже мовилося, реалізація моделі дискреційного контролю доступу пов'язана з наявністю в системі одного з її найважливіших компонентів – монітора безпеки. Це особливий вид суб'єкта, який активується при кожному доступі, в змозі відрізнити легальний доступ від нелегального і не допустити останнього. Моні-

тор безпеки входить до складу диспетчера доступу (SRM), який, згідно з описом, забезпечує також управління ролевим і привілейованим доступом.

Політика безпеки

При оцінці ступеня захищеності операційних систем діє нормативний підхід, відповідно до якого сукупність завдань, що виконуються системою безпеки, повинна задовольняти певним вимогам – їх перелік визначається загальноприйнятими стандартами. Система безпеки ОС Windows відповідає вимогам класу С2 "оранжевої" книги і вимогам стандарту Common Criteria, які складають основу *політики безпеки системи*. Політика безпеки передбачає відповіді на такі питання: яку інформацію захищати, якого роду атаки на безпеку системи можуть бути зроблені, які засоби використовувати для захисту кожного виду інформації.

Вимоги, що висуваються до системи захисту, такі:

1. Кожен користувач повинен бути ідентифікований унікальним вхідним ім'ям і паролем для входу у систему. Доступ до комп'ютера надається лише після аутентифікації. Повинні бути зроблені запобіжні заходи проти спроби застосування фальшивої програми реєстрації (механізм безпечної реєстрації).

2. Система повинна бути в змозі використовувати унікальні ідентифікатори користувачів, щоб стежити за їх діями (управління виборчим або дискреційним доступом). Власник ресурсу (наприклад, файла) повинен мати можливість контролювати доступ до цього ресурсу.

3. Управління довірчими відносинами. Необхідна підтримка наборів ролей (різних типів облікових записів). Окрім того, в системі повинні бути засоби для управління привілейованим доступом.

4. ОС повинна захищати об'єкти від повторного використання. Перед виділенням новому користувачу всі об'єкти, включаючи пам'ять і файли, повинні бути проініційовані.

5. Системний адміністратор повинен мати можливість обліку всіх подій, що належать до безпеки (аудит безпеки).

6. Система повинна захищати себе від зовнішнього впливу або нав'язування, такого, як модифікація завантаженої системи або системних файлів, що зберігаються на диску.

Треба відзначити, що, на відміну від більшості операційних систем, ОС Windows була спочатку спроектована з урахуванням вимог безпеки, і це є її безперечною перевагою. Подивимося тепер, як у рамках даної архітектури забезпечується виконання вимог політики безпеки.

Ролевий доступ. Привілеї

Поняття привілею

З метою гнучкого управління системною безпекою в ОС Windows реалізовано управління довірчими відносинами (trusted facility management), яке вимагає підтримку набору ролей (різних типів облікових записів) для різних рівнів роботи в системі. Треба сказати, що ця особливість системи відповідає вимогам захисту рівня В "оранжевої" книги, тобто жорсткішим вимогам, ніж перераховані в розділі "Політика безпеки". У системі є управління привілейованим доступом, тобто функції адміністрування доступні тільки одній групі облікових записів – Administrators (Адміністратори.).

Відповідно до своєї ролі кожен користувач володіє певними привілеями і правами на виконання різних операцій відносно системи в цілому, наприклад правом на зміну системного часу або правом на створення сторінкового файла. Аналогічні права відносно конкретних об'єктів називаються дозволами. І права, і привілеї призначаються адміністраторами окремим користувачам або групам як частина настройок безпеки. Багато системних функцій (наприклад, LogonUser і InitiateSystemShutdown) вимагають, щоб додаток, який їх викликає, володів відповідними привілеями.

Кожен привілей має два текстові зображення: дружнє ім'я, що відображається в інтерфейсі користувача Windows, і програмне ім'я, використовуване додатками, а також Luid, – внутрішній номер привілею в конкретній системі. Окрім привілеїв, у Windows є близькі до них права облікових записів. Привілеї перераховані у файлі WinNT.h, а права – у файлі NTSecAPI.h із MS Platform SDK. Найчастіше робота з призначенням привілеїв і прав відбу-

ється однаково, хоч і не завжди. Наприклад, функція LookupPrivelegeDisplayName, що перетворює програмне ім'я в дружнє, працює тільки з привілеями.

Нижченаведений перелік програмних імен привілеїв (права відносно системи в даному списку відсутні) облікового запису групи, що відображаються, з адміністративними правами в ОС Windows 2000.

1. SeBackupPrivilege (Архівація файлів і каталогів) .
2. SeChangeNotifyPrivilege (Обхід перехресної перевірки).
3. SeCreatePagefilePrivilege (Створення сторінкового файла).
4. SeDebugPrivilege (Налагодження програм).
5. SeIncreaseBasePriorityPrivilege (Збільшення пріоритету диспетчерування).
6. SeIncreaseQuotaPrivilege (Збільшення квот).
7. SeLoadDriverPrivilege (Завантаження і вивантаження драйверів пристроїв).
8. SeProfileSingleProcessPrivilege (Профілізація одного процесу).
9. SeRemoteShutdownPrivilege (Примусове віддалене завершення).
10. SeRestorePrivilege (Відновлення файлів і каталогів).
11. SeSecurityPrivilege (Управління аудитом і журналом безпеки).
12. SeShutdownPrivilege (Завершення роботи системи).
13. SeSystemEnvironmentPrivilege (Зміна параметрів середовища устаткування).
14. SeSystemProfilePrivilege (Профілізація завантаженості системи).
15. SeSystemtimePrivilege (Зміна системного часу).
16. SeTakeOwnershipPrivilege (Оволодіння файлами або іншими об'єктами).
17. SeUndockPrivilege (Витягування комп'ютера зі стикувального вузла).

Важливо, що навіть адміністратор системи за замовчуванням володіє далеко не всіма привілеями. Це пов'язано з принципом надання мінімуму привілеїв. У кожній новій версії ОС Windows, відповідно до цього принципу, проводиться ревізія переліку користувачів привілеїв, що надаються кожній групі, і загальна тен-

денція полягає в зменшенні їх кількості. З іншого боку, загальна кількість привілеїв у системі росте, що дозволяє проектувати чимраз більш гнучкі сценарії доступу.

Внутрішній номер привілею використовується для специфікації привілеїв, що призначаються суб'єкту, і однозначно пов'язаний з іменами привілею. Наприклад, у файлі WinNT.h це виглядає так:

```
#define SE_SHUTDOWN_NAME      TEXT("SeShutdownPrivilege")
```

Управління привілеями

Призначення і відкликання привілеїв – прерогатива локального адміністратора безпеки LSA (Local Security Authority), тому, щоб програмно призначати і відкликати привілеї, необхідно застосовувати функції LSA. Локальна політика безпеки системи означає наявність набору глобальних відомостей про захист, наприклад, про те, які користувачі мають право на доступ у систему, а також про те, якими вони володіють правами. Тому кажуть, що кожна система, в рамках якої діє сукупність користувачів, що володіють певними привілеями відносно даної системи, є об'єктом політики безпеки. Об'єкт політики використовується для контролю бази даних LSA. Кожна система має тільки один об'єкт політики, який створюється адміністратором LSA під час завантаження і захищений від несанкціонованого доступу з боку додатків.

Використання функцій LSA починається з отримання описувача об'єкта політики (PolicyHandle) за допомогою функції LsaOpenPolicy, яка відкриває описувач об'єкта на локальній або віддаленій машині. Ім'я цільового комп'ютера передається функції як один із параметрів. Після завершення роботи з об'єктом політики необхідно його закрити, викликавши функцію LsaClose(PolicyHandle).

Управління привілеями користувачів включає задачі переліку, задання, видалення, виключення привілеїв і ряд інших. Витягнути перелік привілеїв конкретного користувача можна, наприклад, з маркера доступу процесу, породженого даним користувачем. Там же, в маркері, можна відключити одну або кілька привілеїв.

Про те, як це зробити, буде розказано далі. Проте формувати список привілеїв можна тільки за допомогою LSA API.

Завдання переліку привілеїв облікового запису

Перше завдання – перелік привілеїв даного облікового запису – виконується за допомогою функції LsaEnumerateAccountRights. Ця функція перераховує привілеї користувача із заданим ідентифікатором безпеки Sid відносно системи з об'єктом політики *PolicyHandle*, які передаються їй як параметри.

Зазвичай обліковий запис, якщо не вживати спеціальних заходів, не має привілеїв. Привілеями володіє група, до якої належить даний користувач. Тому не потрібно дивуватися, якщо кількість привілеїв, якими володіє конкретний користувач, у тому числі й адміністратор системи, виявиться таким, що дорівнює 0. Навпаки, набір привілеїв у маркері є сукупністю привілеїв користувача і груп, до яких приписаний даний користувач. З іншого боку, якщо додати привілей користувачу і викликати функцію LsaEnumerateAccountRights, то знов доданий привілей відразу ж буде відмічений, тоді як в маркері доступу процесів даного користувача її не опиниться. Це пов'язано з тим, що такий маркер формується на етапі входу користувача в систему, тому іноді для того, щоб новий привілей користувача потрапив у його маркер доступу, доцільно здійснити повторний вхід у систему.

Дослідження роботи програми переліку привілеїв користувача

Як приклад складіть програму, завдання якої – виведення списку привілеїв поточного користувача.

Дана програма повинна одержувати маркер доступу поточного процесу, витягувати з нього Sid користувача, за допомогою функції LsaOpenPolicy відкривати об'єкт політики безпеки і викликати функцію LsaEnumerateAccountRights для отримання списку привілеїв. Функція LookupPrivilegeDisplayName перетворить програмне ім'я привілею в дружнє ім'я. Для виведення імені привілею на екран російською мовою використовується функція CharToOem. Якщо кількість привілеїв дорівнює нулю, то з урахуванням зауваження, зробленого вище, рекомендується додати ко-

ристувачу, від імені якого запускається програма, однин або кілька привілеїв за допомогою адміністративної консолі управління.

Додавання привілеїв користувачу

Для призначення привілеїв існує функція LsaAddAccountsRights, а для відкликання привілеїв – функція LsaRemoveAccountRights. Функція LsaAddAccountRights призначає один або кілька привілеїв облікового запису. Параметри повторюють параметри функції LsaEnumerateAccountRight, проте цього разу структуру LSA_UNICODE_STRING, яка задає привілей, потрібно сформувати явно. Наприклад, якщо потрібен привілей завершення роботи системи, це можна зробити так:

```
Var pUserRights: PLSA_UNICODE_STRING;
//…
PUserRights[0].Buffer := SE_SHUTDOWN_NAME;
PUserRights[0].Length := lstrlen(PUserRights[0].Buffer) +
                                                    sizeof(WCHAR);
PUserRights[0].MaximumLength := PUserRights[0].Length +
                                                    sizeof(WCHAR);
```

На основі попередньої програми напишіть програму, яка за допомогою функції LsaAddAccountRights розв’язує задачу призначення привілеїв конкретному користувачу.

Аутентифікація користувача. Вхід у систему

У даній частині лабораторної роботи будуть розглянуті питання аутентифікації користувача, системного аудиту, захисту від повторного використання об’єктів і зовнішнього нав’язування, а також можливості тонкої настройки контексту користувача.

Згідно з п. 1 політики безпеки, для доступу до комп’ютера користувач повинен пройти процедуру аутентифікації. Ця процедура ініціюється комбінацією клавіш "ctrl+alt+del". Дана комбінація клавіш, відома як SAS (secure attention sequence), завжди перехоплюється драйвером клавіатури, який викликає при цьому справжню (а не „троянського коня”) програму аутентифікації. Процес користувача не може сам перехопити цю комбінацію клавіш або відмінити її обробку драйвером. Кажучи мовою стандартів, в системі реалізована функціональність шляху довірчих відношень

(trusted path functionality). Дана особливість також відповідає вимогам захисту рівня В „Оранжевої" книги.

Процедурою аутентифікації користувача в системі управляє програма, WinLogon, що є початковою інтерактивною процедурою, яка відображає початковий діалог із користувачем на екрані. Процес WinLogon активно взаємодіє з бібліотекою GINA (Graphic Identification aNd Authentication – графічною бібліотекою ідентифікації і аутентифікації). Бібліотека GINA є замінюваним компонентом, інтерфейс із нею добре документований, тому іноді в додатках, що реалізовують захист, наявна версія GINA, відмінна від оригінальної. Одержавши ім'я і пароль користувача від GINA, WinLogon викликає модуль Lsass для аутентифікації цього користувача. В разі успішного входу в систему Winlogon витягує з реєстру профіль користувача і визначає тип оболонки, що запускається.

Комбінація SAS може бути одержана системою не тільки на етапі реєстрації користувача. Якщо користувач уже увійшов до системи, то після натиснення клавіш "ctrl-alt-del" він дістає можливість: подивитися список активних процесів, ініціювати перезавантаження або вимкнення комп'ютера, змінити свій пароль і заблокувати робочу станцію. У свою чергу, якщо робоча станція заблокована, то після введення SAS користувач має можливість її розблокування. Іноді може бути здійснене примусове виведення користувача із системи з подальшим входом у неї адміністратора.

У процесі аутентифікації викликається системна функція LogonUser, яка, виходячи з імені користувача, його пароля і імені робочої станції або домена, повертає покажчик на маркер доступу користувача. Маркер згодом передається всім дочірнім процесам. При формуванні маркера використовуються ключі SECURITY і SAM реєстру. Перший ключ визначає загальну політику безпеки, а другий ключ містить інформацію про захист для індивідуальних користувачів.

Як самостійне завдання рекомендується здійснити введення комбінації клавіш "ctrl-alt-del" у різних ситуаціях і ініціювати одну з перерахованих вище дій.

Дослідження роботи програми створення нового користувача з правами входу в систему

Напишіть програму, яка створює новий обліковий запис за допомогою функції NetUserAdd, а потім призначає йому право входу в систему SeInteractiveLogonRight шляхом виклику функції LsaAddAccountRights. У разі успішної реалізації зробіть спробу реєстрації в системі як нового користувача.

Дослідження роботи програми отримання маркера доступу на ім'я користувача і його паролю

Як самостійне завдання напишіть програму, яка імітує процедуру входу в систему, тобто одержує новий маркер доступу за допомогою функції LogonUser.

Виявлення вторгнень. Аудит системи захисту

Навіть найкраща система захисту рано чи пізно буде зламана, тому виявлення спроб вторгнення – найважливіше завдання системи захисту. Основним інструментом виявлення вторгнень є запис даних аудиту. Окремі дії користувачів протоколюються, а одержаний протокол використовується для виявлення вторгнень.

Аудит, таким чином, полягає в реєстрації спеціальних даних про різні типи подій, що відбуваються в системі і так чи інакше впливають на стан безпеки комп'ютерної системи. До таких подій зазвичай зараховують такі:

- вхід або вихід із системи;
- операції з файлами (відкрити, закрити, перейменувати, видалити);
- звернення до віддаленої системи;
- зміна привілеїв або інших атрибутів безпеки (режиму доступу, рівня благонадійності користувача і т.п.).

Подія аудиту в ОС Windows може згенеруватися додатком користувача, диспетчером об'єктів або іншим кодом режиму ядра. Щоб ініціювати фіксацію подій, пов'язаних із доступом до об'єкта, необхідно сформувати в дескрипторі безпеки цього об'єкта список SACL, в якому перераховані користувачі, чиї спроби доступу до даного об'єкта підлягають аудиту. Це можна зробити програмно або за допомогою інструментальних засобів ОС.

Для управління файлом журналу безпеки, а також для перегляду і зміни SACL об’єктів, процес повинен володіти привілеєм SeSecurityPrivilege. Процес, що викликає системні записи аудиту, повинен, щоб успішно згенерувати запис аудиту в цьому журналі, володіти привілеєм SeAuditPrivilege.

Як самостійне завдання організуйте аудит доступу до файла за допомогою інструментальних засобів ОС Windows

Для того, щоб примусити систему відстежувати події, пов’язані з доступом до конкретного файла, необхідно зробити таке.

1. Відкрити діалогове вікно "Локальні параметри політики безпеки \ Політика аудиту" (див. рис. 12.2) адміністративної консолі панелі управління.

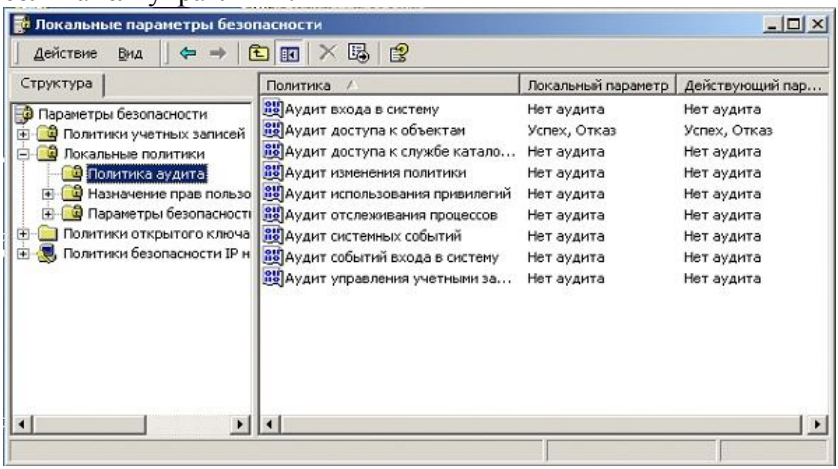


Рис. 12.2. Діалогове вікно "Локальні параметри політики безопасности\ Політика аудиту" адміністративної консолі панелі управління

2. Включити в список перевірки потрібну сукупність подій, наприклад аудит доступу до об’єктів.

3. За допомогою програми Windows Explorer вибрати файл для аудиту. Викликати панель фіксації подій, які пов’язані з файлом, що підлягає аудиту. Для цього за допомогою миші в контекстному меню відкрити вікно "Властивості", вибрати вкладку "Безпека", потім, натиснувши кнопку "Додатково", вибрати "Аудит".

Потім за допомогою кнопки "Додати" вибрати користувача, дії якого підлягають аудиту, і перелік подій, що підлягають аудиту (див. рис. 12.3).

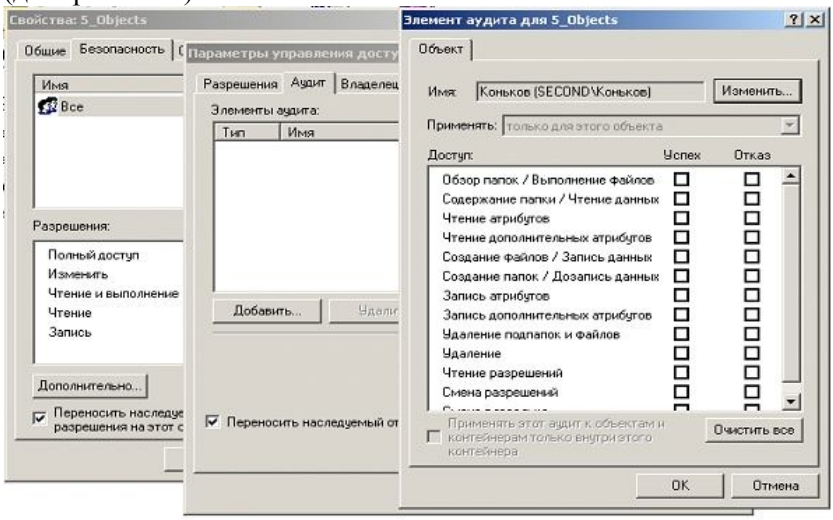


Рис. 12.3. Діалогове вікно установки подій, що підлягають аудиту для конкретного файла

4. У випадку успіху спробувати здійснити звернення до даного файла. Всі обумовлені дії відносно файла повинні знайти віддзеркалення в журналі подій безпеки. Для перегляданння журналу подій потрібно вибрати вікно "Перегляданння подій" через іконку "Адміністрування" панелі управління.

5. У випадку відмови перевірити наявність у користувача (у тому числі й у адміністратора), від імені якого проводиться експеримент, привілеї "Створення журналів безпеки" і, в разі її відсутності, призначити її користувачу за допомогою консолі "Призначення прав користувачам".

Як уже мовилося, список SACL, що входить до складу дескриптора захисту об’єкта, можна формувати і модифікувати програмними засобами. Це можна зробити за допомогою функції SetSecurityInfo, яка є зворотною вже знайомій нам функції GetSecurutyInfo і містить той же набір параметрів. Оскільки йдеться про аудит, то параметр SecurityInfo, який специфікує

компонент дескриптора захисту, що підлягає зміні, повинен включати значення SACL_SECURITY_INFORMATION.

Неприпустимість повторного використання об'єктів

Згідно з п. 4 політики безпеки, ОС повинна захищати об'єкти від повторного використання. Перед виділенням новому користувачу всі об'єкти, включаючи пам'ять і файли, повинні бути проініційовані. Контроль повторного використання об'єкта призначений для запобігання спробам незаконного отримання конфіденційної інформації, залишки якої могли зберегтися в деяких об'єктах, що раніше використалися і звільнених іншим користувачем.

Безпека повторного застосування повинна гарантуватися для областей оперативної пам'яті (зокрема, для буферів з образами екрана, розшифрованими паролями і т.п.), для дискових блоків і магнітних носіїв у цілому. Очищення повинне проводитися шляхом запису маскуючої інформації в об'єкт при його звільненні (перерозподілі).

У ОС Windows гарантується безпека повторного використання областей фізичної пам'яті. Якщо процесу користувача знадобилася вільна сторінка пам'яті, вона може бути виділена тільки зі списку обнулених сторінок бази даних PFN (див. лабораторну роботу, пов'язану з роботою менеджера пам'яті). Якщо цей список порожній, сторінка береться зі списку вільних сторінок і заповнюється нулями. Якщо і цей список порожній, диспетчер пам'яті витягує сторінку зі списку простоюючих (standby) сторінок і зануляє її. Незанулена сторінка передається тільки для відображення проектованого файла. В цьому випадку фрейм незануленої сторінки ініціалізувався даними з диска.

Що стосується повторного використання вмісту файлів, добре поміркувавши, стає зрозуміло, що у звичайного зловмисника практично відсутні механізми отримання інформації, що зберігається в дискових блоках віддалених файлів. Виділення нових дискових блоків здійснюється тільки для операцій запису на диск даних із заздалегідь занулених сторінок пам'яті. Зрозуміло, за наявності адміністративних прав і фізичного доступу до комп'ютера можна здійснити злам системи захисту, але в подіб-

них ситуаціях для отримання конфіденційної інформації існують простіші способи.

Захист від зовнішнього нав'язування

Відповідно до політики безпеки операційна система повинна захищати себе від зовнішнього впливу або нав'язування, такого як модифікація завантаженої системи або системних файлів, що зберігаються на диску. Щоб задовольнити вимогам політики безпеки, в ОС Windows вбудовані засоби захисту файлів (Windows File Protection, WFP), які захищають системні файли навіть від змін із боку користувача з адміністративними правами. У основі захисту лежать засоби фіксації змін у системних файлах. Дана міра, безумовно, підвищує стабільність системи.

Згідно з документацією, моніторингу і захисту підлягають усі файли, що поставляються у складі ОС, з розширеннями sys, dll, exe і ocx, а також деякі шрифти TrueType (Micros.ttf, Tahoma.ttf і Tahomabd.ttf). Якщо з'ясовується, що файл підмінено, він замінюється копією з каталога %systemroot%\system32\dllcache, на який указує один із записів у реєстрі. Якщо розмір місця, виділеного для цього каталогу, не достатній, то в ньому зберігаються копії не всіх системних файлів. У тих випадках, коли робиться спроба видалення системного файла і при цьому його не опиняється в каталозі dllcache, система намагається відновити його з установочного компакт-диска ОС Windows або з мережних ресурсів.

Окрім того, адміністратор системи за допомогою утиліти Sfc.exe (system file checker) може здійснити перевірку коректності версії всіх системних файлів. Відомості про файли з некоректним номером версії заносяться в протокол. Коректність системних файлів перевіряється за допомогою механізму електронного підпису і до непідписаних файлів слід ставитися з обережністю. Для перевірки підпису і виявлення непідписаних файлів служить штатна утиліта SigVerif.exe.

Як самостійне завдання спробуйте видалити який-небудь системний файл із каталогу Windows\system32, наприклад sol.exe (гра "косинка").

Маркер доступу. Контекст користувача

Як уже мовилося, найбільш важливою характеристикою суб'єкта є маркер доступу. Зазвичай маркер створюється при інтерактивному вході користувача в систему і зберігає відомості про контекст користувача. Спочатку маркер зв'язується з процесом-оболонкою Windows Explorer, а потім усі процеси, породжені користувачем під час сеансу роботи, одержують дублікат даного маркера. Важливо розуміти, що самостійно створити маркер додаток користувача не може. Це може зробити тільки служба Lsass.

Раніше були розглянуті характеристики маркера як однієї з ключових ланок у системі контролю доступу. В цьому розділі описані можливості тоншої настройки контексту користувача за допомогою функцій, зорієнтованих на роботу з маркером доступу відповідно до принципу мінімуму привілеїв. Принцип мінімальних привілеїв рекомендує виконання всіх операцій із мінімальними привілеями, необхідними для досягнення результату. Це дозволяє зменшити втрати від спроб навмисного збитку й уникнути випадкових втрат даних. Наприклад, користувачу не рекомендується реєструватися як адміністратор системи без необхідності. (В крайньому випадку, можна вдатися до послуг штатної утиліти runas, яка дозволяє запускати додатки від імені іншого облікового запису.)

Для безпечної роботи в дусі принципу мінімуму привілеїв ОС Windows підтримує механізми створення маркерів з обмеженими привілеями і запозичення маркера. Перший дозволяє вилучити з маркера певні привілеї, наявність яких при виконанні даної операції не потрібна, а другий дозволяє запускати додатки від імені іншого облікового запису. API системи дозволяє впливати на перелік діючих привілеїв маркера, дублювати маркер, щоб його міг запозичувати інший процес, розв'язувати проблеми перевтілення і створення обмежених маркерів

Зчитування відомостей про привілеї користувача з маркера доступу

У попередній лабораторній роботі описані привілеї користувача (розділ "ролевий доступ") і основні функції для зчитування

інформації з маркера (OpenProcessToken, GetTokenInformation) у розділі "суб'єкти". Для отримання списку привілеїв із маркера за допомогою функції GetTokenInformation потрібно для параметра TokenInformationClass, вибрати значення TokenPrivileges. Раніше ми вже одержували перелік привілеїв облікового запису. Перелік привілеїв, що входять до складу маркера, істотно ширше, оскільки включає привілеї користувача і привілеї груп, до складу яких входить користувач.

Дослідження роботи програми отримання списку привілеїв із маркера доступу процесу

Як самостійний приклад складіть програму, завдання якої – отримання переліку привілеїв користувача і його груп із маркера доступу даного процесу.

Суть роботи складеної програми – одержати описувач поточного процесу за допомогою функції OpenProcessToken і застосувати функцію GetTokenInformation для витягування з нього списку привілеїв. Функція LookupPrivilegeName дозволяє одержати програмне ім'я привілею по її локальному ідентифікатору (Luid), а функція LookupPrivilegeDisplayName перетворить програмне ім'я привілею в дружнє ім'я.

Включення і відключення привілеїв у маркері

Велику частину інформації в маркері змінювати не можна. Наприклад, не можна ні додати, ні видалити привілей. Можливість додавання привілеїв у маркер була б "глибоким підкопом під систему захисту Windows". Останнє означає, що зміна привілеїв суб'єкта – прерогатива підсистеми локальної авторизації (LSA) і повинна здійснюватися централізовано тільки за допомогою функцій сімейства LSA.

Отже, сформувати список привілеїв у маркері не можна, зате їх можна відключати і включати. Для включення і відключення певних привілеїв служить функція AdjustTokenPrivileges.

Перевтілення

У ОС Windows є цікаві можливості виконання коду з маркером, відмінним від маркера даного процесу. Це, наприклад, істотно для серверів, які повинні виступати від імені і з привілеями клієнтів. Одна з таких можливостей – запуск нового процесу з

вказаним маркером за допомогою функції CreateProcessAsUser. Ця функція є аналогом функції CreateProcess, за винятком параметра hToken, який указує на маркер, що визначає контекст користувача нового процесу.

Інший спосіб виконати код із запозиченим маркером – здійснити *перевтілення (impersonation)*. Перевтілення означає, що один із потоків процесу функціонує з маркером, відмінним від маркера поточного процесу. Зокрема, клієнтський потік може передати свій маркер доступу серверному потоку, щоб сервер міг дістати доступ до захищених файлів і інших об'єктів від імені клієнта. Згодом потік може повернутися в нормальний стан, тобто використовувати маркер процесу.

Windows не дозволяє серверам виступати в ролі клієнтів без їх відома. Щоб уникнути цього, клієнтський процес може обмежити рівень імперсонації. Тому для участі в перевтіленні маркер повинен мати відповідний рівень перевтілення. Цей рівень визначається параметром маркера TokenImpersonationLevel і може бути запитаний функцією GetTokenInformation. Для процесу перевтілення важливо, щоб цей параметр мав значення не нижче за SecurityImpersonation, що означає можливість імперсонації на локальній машині. Значення SecurityDelegation дозволяє серверному процесу виступати від імені клієнта як на локальному, так і на віддаленому комп'ютері. Останнє значення має відношення до делегування, яке є природним розвитком перевтілення, але працює тільки за наявності домена і функціонуванні активного каталогу. За замовчуванням встановлюється рівень SecurityImpersonation.

Маркер перевтілення зазвичай створюють шляхом дублювання існуючого маркера і додання йому потрібних прав доступу і рівня імперсонації. Це можна зробити за допомогою функції DuplicateTokenEx. Власне, перевтілення здійснюється за допомогою функції ImpersonateLoggedOnUser. Щоб повернутися в початковий стан, потік повинен викликати функцію RevertToSelf.

Важливо також не забувати, що коли перевтілений потік здійснює доступ до об'єкта, то решта потоків процесу, навіть не перевтілених, також має до нього доступ. Подібні ситуації вимагають

ретельного аналізу, оскільки тут можуть виникати важко відстежувані помилки.

Дослідження роботи програми запозичення маркера доступу й отримання з нього потрібної інформації

Як самостійну вправу рекомендується написати програму, завдання якої – створити маркер доступу з потрібним рівнем перевтілення для вказаного облікового запису, і передати його поточному процесу. Потім для контролю потрібно вивести на екран основні характеристики цього (що вже став поточним) маркера, наприклад перелік привілеїв, після чого повернути значення поточного маркера до початкового і ще раз витягнути з нього привілеї.

Інші можливості настройки контексту користувача

У ОС Windows є й інші можливості настройки контексту користувача відповідно до складних вимог захисту з урахуванням принципу мінімуму привілеїв. Як приклад можна навести схему контролю доступу за допомогою маркерів, які називаються обмеженими. Обмежені маркери створюються функцією CreateRestrictedToken. У цей маркер у момент його створення можна внести такі зміни: видалити привілеї, відключити деякі SID-ідентифікатори і додати "обмежені" SID'и облікових записів. Останні просто додають ряд нових перевірок до тих, що вже існують при організації доступу до об'єкта. Обмежені маркери зручні, коли додаток підміняє клієнта при виконанні коду, здатного завдати збитку системі.

Висновок

У даній лабораторній роботі описана структура менеджера безпеки ОС Windows. Система захисту даних повинна задовольняти вимогам, сформульованим у ряді нормативних документів, які визначають політику безпеки. Далі в роботі описані можливості настройки привілеїв облікового запису. Підтримка моделі ролевого доступу пов'язана із завданнями переліку, додавання і відгуку привілеїв користувача і відключення привілеїв у маркері доступу суб'єкта.

Відповідно до політики безпеки, в системі реалізовані: аутентифікація користувачів, аудит і захист від повторного використ-

тання об’єктів. Успішна аутентифікація закінчується формуванням маркера доступу, який передається всім процесам користувача протягом сеансу роботи. Для спостереження за діями користувачів підтримується журнал, де можна фіксувати всі події, які можуть вплинути на стан безпеки системи. Зануління сторінок пам’яті перед виділенню їх процесу покликане не допустити зчитування інформації, що залишилася від їх колишнього власника. У ОС Windows є захист від модифікації системних файлів, яка базується на фіксації змін у файлах і заміні спотворених системних файлів їх резервною копією. Для тонкої настройки контексту користувача відповідно до складних сценаріїв захисту підтримується механізм перевтілення.

Практична частина

1. Складіть програми, рекомендовані як самостійні вправи. Запустіть програми та переконайтеся, що вони працюють правильно (згідно з описом алгоритму їх роботи).

4. Оформіть звіт із виконаної лабораторної роботи, який повинен містити текст програм, демонстрацію результатів роботи, та дайте відповіді на контрольні запитання.

Контрольні запитання та завдання

1. Опишіть структуру системи безпеки ОС Windows.
2. Яке призначення політики безпеки?
3. Розкрийте зміст поняття привілею.
4. Наведіть перелік програмних імен привілеїв облікового запису групи.
5. Які API-функції управління привілеями ви знаєте?
6. Розкрийте зміст поняття аутентифікації користувача.
7. Які інструменти виявлення вторгнень в ОС ви знаєте?
8. Обґрунтуйте необхідність захисту від повторного використання об’єктів. Наведіть методи захисту.
9. Обґрунтуйте необхідність захисту від зовнішнього нав’язування. Наведіть методи захисту.
10. Розкрийте зміст поняття перевтілення. Для чого воно використовується?

Список літератури

1. Шеховцов В.А. Операційні системи. – К.: Видавнича група BHV, 2005. – 576 с.
2. Саймон Р. Windows 2000 API. – СПб.: Питер, 2002. – 1085 с.
3. Таненбаум Э., Вудхалл А. Операционные системы: разработка и реализация. – СПб.: Питер, 2006. – 576 с.
4. Олифер В.Г., Олифер Н.А. Сетевые операционные системы. – СПб.: Питер, 2002. – 544 с.
5. Таненбаум Э. Современные операционные системы. – 2-е изд. – СПб.: Питер, 2002. – 1040 с.
6. Щупак Ю. А. Win32 API. Эффективная разработка приложений. – СПб.: Питер, 2007. – 572 с.
7. Побегайло А. П. Системное программирование в Windows. – СПб.: БХВ-Петербург, 2006. – 1056 с.
8. Вильямс А. Системное программирование в Windows для профессионалов. – СПб.: Питер, 2007. – 572 с.
9. Илюшкин Б. И. Операционные системы. Процессы и потоки: Учебное пособие. – СПб.: СЗТУ, 2005. – 103 с.
10. Румянцев П. В. Азбука программирования в Win32 API. – СПб.: Питер, 2000. – 310 с.
11. Безбогов, А.А. Безопасность операционных систем: Учебное пособие. – М.: Машиностроение-1, 2007. – 220 с.
12. Рощин А.В. Операционные системы. Часть 1. Основы управления ресурсами: Учебное пособие. – М.: МГУПИ, 2007. – 119 с.: ил.
13. Харт Д. М. Системное программирование в среде Windows – 3-е изд. – М.: Издательский дом «Вильямс», 2005. – 592 с.
14. Румянцев П.В. Работа с файлами в WIN 32 API. – СПб.: Питер, 2005. – 197 с.
15. Финогенов К.Г. WIN 32 Основы программирования. 2-е изд. – М.: Диалог МИФИ, 2006. – 416 с.

Навчальне видання

Операційні системи
Методичні рекомендації до лабораторних робіт

Укладач *Жихаревич Володимир Вікторович*

Відповідальний за випуск *Остапов С.Е.*
Літературний редактор *Колодій О.В.*

Друкарня видавництва “Рута”
Чернівецького національного університету ім. Ю. Федьковича
58012, Чернівці, вул. Коцюбинського, 2